Ontologies in Quantum

Modeling Relationships That Are Resilient and Fast

Version 1.2.2-SNAPSHOT, 2025-10-19T00:51:33Z

Table of Contents

Chapter 1. Ontologies in Quantum: Modeling Relationships That Are Resilient and Fast

Looking for the short implementation plan? See PROPOSAL.md at the repository root for a concise module-by-module checklist.

This section explains what an ontology is, how it differs from a traditional object model, and how the Quantum Ontology modules make it practical to apply ontology ideas to your domain models and queries. It also contrasts ontology-driven relationships with direct object references (for example, using @Reference or EntityReference).

1.1. What is an Ontology?

In software terms, an ontology is a formal, explicit specification of concepts and their relationships.

- Concepts (Classes): Named categories/types in your domain. Concepts can form taxonomies (is-a hierarchies), be declared disjoint, or be equivalent.
- Relationships (Properties): Named relationships between entities. Properties can have a domain (applies to X) and a range (points to Y). They may be inverse or transitive.
- Axioms (Rules): Constraints and entailment rules, including property chains such as: if (A --p-→ B) and (B --q-→ C) then we infer (A --r-→ C).
- Inference: The process of deriving new facts (types, labels, edges) that were not explicitly stored but follow from axioms and known facts.

An ontology is not the data; it is the schema plus logic that gives your data additional meaning and enables consistent, automated inferences.

1.2. Ontology vs. Object Model

A conventional object model focuses on concrete classes, fields, and direct references between objects at implementation time. An ontology focuses on semantic types and relationships, with explicit rules that can derive new knowledge independent of how objects are instantiated.

Key differences: - Purpose - Object model: Encapsulate data and behavior for application code generation and persistence. - Ontology: Encode shared meaning, constraints, and inference rules that remain stable as implementation details change. - Relationship handling - Object model: Typically uses direct references or foreign keys; traversals are hard-coded and fragile to change. - Ontology: Uses named predicates (properties) and can infer additional relationships by rules (property chains, inverses, transitivity). - Polymorphism and evolution - Object model: Polymorphism requires class inheritance in code; cross-cutting categories are awkward to add later. - Ontology: Entities can have multiple types/labels at once. New concepts and properties can be introduced without breaking existing data. - Querying - Object model: Queries couple to concrete classes and field paths; changes force query rewrites. - Ontology: Queries target semantic relationships; reasoners can materialize edges that queries reuse, decoupling queries from implementation details.

1.3. Why prefer Ontology-driven relationships over @Reference/EntityReference

Direct references (@Reference or custom EntityReference) are simple to start but become restrictive as domains grow: - Tight coupling: Code and queries couple to concrete field paths (customer.primaryAddress.id), making refactors risky. - Limited expressivity: Hard to encode and reuse higher-order relationships (e.g., "partners of my supplier's parent org"). - Poor polymorphism: References point to one collection/type; accommodating multiple target types requires extra code. - Performance pitfalls: Deep traversals cause extra queries, N+1 selects, or complex $lookup joins.

Ontology-driven edges address these issues: - Decoupling via predicates: Use named predicates (e.g., hasAddress, memberOf, supplies) that remain stable while internal object fields change. - Inference for reachability: Property chains can materialize implied links (A --p-→ B & B --q-→ C ⇒ A --r-→ C), avoiding runtime multi-hop traversals. - Polymorphism-first: A predicate can connect heterogeneous types; type inferences (domain/range) remain consistent. - Query performance: Pre-materialized edges allow single-hop, index-friendly queries (in or eq filters) instead of ad-hoc multi-collection traversals. - Resilience to change: You can add or modify rules without rewriting data structures or touching referencing fields across models.

1.4. How Quantum supports Ontologies

Quantum provides three cooperating modules that make ontology modeling practical and fast:

- quantum-ontology-core (package com.e2eq.ontology.core)
- OntologyRegistry: Holds the TBox (terminology) of your ontology.
- ClassDef: Concept names and relationships (parents, disjointWith, sameAs).
- PropertyDef: Property names with optional domain, range, inverse flags, and transitivity.
- PropertyChainDef: Rules that define multi-hop implications (chains → implied property).
- TBox: Container for classes, properties, and property chains.
- Reasoner interface and ForwardChainingReasoner: Given an entity snapshot and the registry, computes inferences:
- New types/labels to assert on entities.
- New edges to add (implied by property chains, inverses, or other rules).
- quantum-ontology-mongo (package com.e2eq.ontology.mongo)
- EdgeDao: A thin DAO around an edges collection in Mongo. Each edge contains tenantId, src, predicate p, dst, inferred flag, provenance, and timestamp.
- OntologyMaterializer: Runs the Reasoner for an entity snapshot and upserts the inferred edges, so queries can be rewritten to simple in/in eq filters.
- quantum-ontology-policy-bridge (package com.e2eq.ontology.policy)
- ListQueryRewriter: Takes a base query and rewrites it using the EdgeDao to filter by the set of source entity ids that have a specific predicate to a given destination.
- This integrates ontology edges with RuleContext or policy decisions: policy asks for entities

related by a predicate; the rewriter converts that into an efficient Mongo query.

These modules let you define your ontology (core), materialize derived relations (mongo), and leverage them in access and list queries (policy bridge).

1.5. Modeling guidance: from object fields to predicates

- Name relationships explicitly
- Define clear predicate names (hasAddress, memberOf, supplies, owns, assignedTo). Avoid encoding relationship semantics in field names only.
- Keep object model minimal and flexible
- Store lightweight identifiers (ids) as needed, but avoid deeply nested reference graphs that encode traversals in code.
- Model polymorphic relationships
- Prefer predicates that naturally connect multiple possible types (e.g., assignedTo can target User, Team, Bot) and rely on ontology type assertions to constrain where needed.
- Use property chains for common paths
- If business logic often traverses A → B → C, define a chain p□q ⇒ r and materialize r for faster queries and simpler policies.
- Capture inverses and transitivity
- For natural inverses (parentOf □ childOf) or transitive relations (partOf, locatedIn), define them in the ontology so edges and queries stay consistent.
- Keep provenance
- Record why an edge exists (prov.rule, prov.inputs) so you can recompute, audit, or retract when inputs change.

1.6. Querying with ontology edges vs direct references

- Direct reference example (fragile/slow)
- Query: "Find Orders whose buyer belongs to Org X or its parents."
- With @Reference: requires joining Order → User → Org and recursing org.parent; costly and tightly coupled to fields.
- Ontology edge example (resilient/fast)
- Define predicates: placedBy(order, user), memberOf(user, org), ancestorOf(org, org). Define chain placedBy □ memberOf ⇒ placedInOrg.
- Materialize edges: (order --placedInOrg-→ org). Also make ancestorOf transitive.
- Query becomes: where order._id in EdgeDao.srcIdsByDst(tenantId, "placedInOrg", orgX).
- With transitivity, you can precompute ancestor closure or add a chain placedInOrg □ ancestorOf ⇒ placedInOrg to include parents automatically.

1.7. Migration: from @Reference to ontology edges

- Start by introducing predicates alongside existing references; do not remove references immediately.
- Materialize edges for hot read paths; keep provenance so you can reconstruct.
- Gradually update queries (list screens, policy filters) to use ListQueryRewriter with EdgeDao instead of deep traversals or $lookup.
- Once stable, you can simplify models by removing rigid reference fields where unnecessary and rely on edges for read-side composition.

1.8. Performance and operational notes

- Indexing: Create compound indexes on edges: (tenantId, p, dst) and (tenantId, src, p) to support both reverse and forward lookups.
- Write amplification vs read wins: Materialization adds write work, but dramatically improves read latency and simplifies queries.
- Consistency: Re-materialize edges on relevant entity changes (source, destination, or intermediate) using OntologyMaterializer.
- Multi-tenancy: Keep tenantId in the edge key and filters; the provided EdgeDao methods include tenant scoping.

1.9. How this integrates with Functional Areas/Domains

- Functional domains often map to concept clusters in the ontology. Use @FunctionalMapping to aid discovery and apply policies per area/domain.
- Policies can refer to relationships semantically ("hasEdge placedInOrg OrgX") and rely on the policy bridge to turn this into efficient data filters.

1.10. Summary

- Ontology-powered relationships provide a stable, semantic layer over your object model.
- The Quantum Ontology modules let you define, infer, and query these relationships efficiently on MongoDB.
- Compared with direct @Reference/EntityReference, ontology edges are more expressive, resilient to change, and typically faster for complex list/policy queries once materialized.

Chapter 2. Concrete example: Sales Orders, Shipments, and evolving to Fulfillment/Returns

This example shows how to use an ontology to model relationships around Orders, Customers, and Shipments, and how the model can evolve to include Fulfillment and Returns without breaking existing queries. We will:

- Define core concepts and predicates.
- Add property chains that materialize implied relationships for fast queries.
- Show how queries are rewritten using edges instead of deep object traversals.
- Evolve the model to support Fulfillment and Returns with minimal changes.

Core concepts (classes)

- Order, Customer, Organization, Shipment, Address, Region
- Later evolution: FulfillmentTask, FulfillmentUnit, ReturnRequest, ReturnItem, RMA

Key predicates (relationships)

- placedBy(order, customer): who placed the order
- memberOf(customer, org): a customer belongs to an organization (or account)
- orderHasShipment(order, shipment): outbound shipment for the order
- shipsTo(shipment, address): shipment destination
- locatedIn(address, region): address is located in a Region
- ancestorOf(org, org): organizational ancestry (transitive)

Property chains (implied relationships)

- placedBy memberOf ⇒ placedInOrg
- If (order --placedBy-→ customer) and (customer --memberOf-→ org), then infer (order --placedInOrg-→ org)
- orderHasShipment shipsTo ⇒ orderShipsTo
- If (order --orderHasShipment-→ shipment) and (shipment --shipsTo-→ address), infer (order --orderShipsTo-→ address)
- orderShipsTo locatedIn ⇒ orderShipsToRegion
- If (order --orderShipsTo-→ address) and (address --locatedIn-→ region), infer (order --orderShipsToRegion-→ region)
- placedInOrg ancestorOf ⇒ placedInOrg
- Makes placedInOrg resilient to org hierarchy changes (ancestorOf is transitive). This is a common “closure” trick: re-assert the same predicate via chain to absorb hierarchy.

A minimal Java-style snippet to define this TBox

```
import java.util.*;
import com.e2eq.ontology.core.OntologyRegistry;
import com.e2eq.ontology.core.OntologyRegistry.*;

Map<String, ClassDef> classes = Map.of(
  "Order", new ClassDef("Order", Set.of(), Set.of(), Set.of()),
  "Customer", new ClassDef("Customer", Set.of(), Set.of(), Set.of()),
  "Organization", new ClassDef("Organization", Set.of(), Set.of(), Set.of()),
  "Shipment", new ClassDef("Shipment", Set.of(), Set.of(), Set.of()),
  "Address", new ClassDef("Address", Set.of(), Set.of(), Set.of()),
  "Region", new ClassDef("Region", Set.of(), Set.of(), Set.of())
);

Map<String, PropertyDef> props = Map.of(
  "placedBy", new PropertyDef("placedBy", Optional.of("Order"), Optional.of("Customer
"), false, Optional.empty(), false),
  "memberOf", new PropertyDef("memberOf", Optional.of("Customer"), Optional.of(
"Organization"), false, Optional.empty(), false),
  "orderHasShipment", new PropertyDef("orderHasShipment", Optional.of("Order"),
Optional.of("Shipment"), false, Optional.empty(), false),
  "shipsTo", new PropertyDef("shipsTo", Optional.of("Shipment"), Optional.of("Address
"), false, Optional.empty(), false),
  "locatedIn", new PropertyDef("locatedIn", Optional.of("Address"), Optional.of(
"Region"), false, Optional.empty(), false),
  "ancestorOf", new PropertyDef("ancestorOf", Optional.of("Organization"), Optional.
of("Organization"), false, Optional.empty(), true), // transitive
  // implied predicates (no domain/range required, but you may add them for
validation)
  "placedInOrg", new PropertyDef("placedInOrg", Optional.of("Order"), Optional.of(
"Organization"), false, Optional.empty(), false),
  "orderShipsTo", new PropertyDef("orderShipsTo", Optional.of("Order"), Optional.of(
"Address"), false, Optional.empty(), false),
  "orderShipsToRegion", new PropertyDef("orderShipsToRegion", Optional.of("Order"),
Optional.of("Region"), false, Optional.empty(), false)
);

List<PropertyChainDef> chains = List.of(
  new PropertyChainDef(List.of("placedBy", "memberOf"), "placedInOrg"),
  new PropertyChainDef(List.of("orderHasShipment", "shipsTo"), "orderShipsTo"),
  new PropertyChainDef(List.of("orderShipsTo", "locatedIn"), "orderShipsToRegion"),
  new PropertyChainDef(List.of("placedInOrg", "ancestorOf"), "placedInOrg")
);

OntologyRegistry.TBox tbox = new OntologyRegistry.TBox(classes, props, chains);
OntologyRegistry registry = OntologyRegistry.inMemory(tbox);
```

Materializing edges for an Order

- Explicit facts for order O1:
- O1 placedBy C9
- C9 memberOf OrgA
- O1 orderHasShipment S17
- S17 shipsTo Addr42
- Addr42 locatedIn RegionWest
- OrgA ancestorOf OrgParent
- Inferred edges after running the reasoner for O1's snapshot:
- O1 placedInOrg OrgA
- O1 placedInOrg OrgParent (via closure with ancestorOf)
- O1 orderShipsTo Addr42
- O1 orderShipsToRegion RegionWest

How queries become simple and fast

- List Orders for Organization OrgParent (including children):
- Instead of joining Order → Customer → Org and recursing org.parent, run a single filter using materialized edges.

```
import com.mongodb.client.model.Filters;
import org.bson.conversions.Bson;
import com.e2eq.ontology.policy.ListQueryRewriter;

Bson base = Filters.eq("status", "OPEN");
Bson rewritten = rewriter.rewriteForHasEdge(base, tenantId, "placedInOrg", "OrgParent
");
// Use rewritten in your Mongo find
```

- List Orders shipping to RegionWest:

```
Bson rewritten2 = rewriter.rewriteForHasEdge(Filters.empty(), tenantId,
"orderShipsToRegion", "RegionWest");
```

Why this is resilient

- If tomorrow Customer becomes AccountContact and the organization model gains Divisions and multi-parent org graphs, you only adjust predicates and chains.
- Queries that rely on placedInOrg or orderShipsToRegion remain unchanged and fast, because edges are re-materialized by OntologyMaterializer.

Evolving the model: add Fulfillment

New concepts

- FulfillmentTask: a unit of work to pick/pack/ship order lines
- FulfillmentUnit: a logical grouping (e.g., wave, tote, parcel)

New predicates

- fulfills(task, order)
- realizedBy(order, fulfillmentUnit)
- taskProduces(task, shipment)

New chains (implied)

- fulfills ⇒ derived edge from task to order; combine with taskProduces to connect order to shipment without touching Order fields:
- fulfills taskProduces ⇒ orderHasShipment
- realizedBy orderHasShipment ⇒ fulfilledByUnit
- If (order --realizedBy-→ fu) and (order --orderHasShipment-→ s) ⇒ (fu --fulfillsShipment-→ s) or simply (order --fulfilledByUnit-→ fu)

These chains let you introduce warehouse concepts without changing how UI filters orders by organization or ship-to region. Existing queries still operate via placedInOrg and orderShipsToRegion.

Evolving further: add Returns

New concepts

- ReturnRequest, ReturnItem, RMA

New predicates

- hasReturn(order, returnRequest)
- returnFor(returnItem, order)
- returnRma(returnRequest, rma)

New chains (implied)

- hasReturn ⇒ openReturnOnOrg via placedInOrg:
- hasReturn placedInOrg ⇒ returnPlacedInOrg
- returnFor orderShipsToRegion ⇒ returnShipsToRegion

Example queries with new capabilities

- List Orders with open returns in OrgParent:

```
Bson r = rewriter.rewriteForHasEdge(Filters.empty(), tenantId, "returnPlacedInOrg",
"OrgParent");
```

- List Returns associated to Orders shipping to RegionWest:

```
Bson r2 = rewriter.rewriteForHasEdge(Filters.empty(), tenantId, "returnShipsToRegion",
"RegionWest");
```

Comparison with direct references (@Reference/EntityReference)

- With direct references you would encode fields like Order.customer, Order.shipments, Shipment.address, Address.region and then implement multi-hop traversals in code or $lookup pipelines, rewriting them whenever you add Fulfillment or Returns.
- With ontology edges, you keep predicates stable and add property chains. Existing list and policy queries keep working and typically become faster due to single-hop filters on an indexed edges collection.

Operational tips for this scenario

- Ensure EdgeDao has indexes on (tenantId, p, dst) and (tenantId, src, p).
- Use OntologyMaterializer when Order, Shipment, Customer, Address, or org hierarchy changes to keep edges fresh.
- Keep provenance in edge.prov (rule, inputs) so you can recompute or retract edges when source data changes.

Chapter 3. Integrating Ontology with Morphia, Permissions, and Multi-tenancy

This section focuses on integration and developer experience: how ontology edges flow into Morphia-based repositories and the permission rule language, while remaining fully multi-tenant and secure.

3.1. Big picture: where ontology fits

- Write path (materialization):
- Your domain code persists entities with minimal direct references.
- An OntologyMaterializer runs when entities change to derive and upsert edges into the edges collection (per tenant).
- Policy path (authorization and list filters):
- The permission rule language evaluates the caller's SecurityContext/RuleContext and produces logical filters.
- When a rule asks for a semantic relationship (hasEdge), we use ListQueryRewriter + EdgeDao to translate that into efficient Mongo filters over ids.
- Read path (queries):
- Morphia repos apply the base data-domain filters and the rewritten ontology constraint to queries, producing fast lists without deep joins.

3.2. Rule language: add hasEdge()

We introduce a policy function/operator to reference ontology edges directly from rules:

- Signature: hasEdge(predicate, dstIdOrVar)
- predicate: String name of the ontology predicate (e.g., "placedInOrg", "orderShipsToRegion").
- dstIdOrVar: Either a concrete id/refName or a variable resolved from RuleContext (e.g., principal.orgRefName, request.region).
- Semantics: The rule grants/filters entities for which an edge (tenantId, src = entity._id, p = predicate, dst = resolvedDst) exists.
- Composition: hasEdge can be combined with existing rule clauses (and/or/not) and other filters (states, tags, ownerId, etc.).

Example rule snippets (illustrative):

- Allow viewing Orders in the caller's org (including ancestors via ontology closure):
- allow VIEW Order when hasEdge("placedInOrg", principal.orgRefName)
- Restrict list to Orders shipping to a region chosen in request:
- allow LIST Order when hasEdge("orderShipsToRegion", request.region)

Under the hood, policy evaluation uses ListQueryRewriter.rewriteForHasEdge(...), which converts hasEdge into a set of source ids and merges that with the base query.

3.3. Passing tenantId correctly

- Always resolve tenantId from RuleContext/SecurityContext (the same source your repos use for realm/database selection).
- EdgeDao and ListQueryRewriter already accept tenantId; never cross tenant boundaries when reading edges.
- Index recommendation (per tenant):
- (tenantId, p, dst)
- (tenantId, src, p)

3.4. Morphia repository integration patterns

The goal is zero-friction usage in existing repos without invasive changes.

Option A: Apply ontology constraints in code paths that already construct BSON filters.

- If your repo method builds a Bson filter before calling find(), wrap it through rewriter:

```
Bson base = Filters.and(existingFilters...);
Bson rewritten = hasEdgeRequested
  ? rewriter.rewriteForHasEdge(base, tenantId, predicate, dst)
  : base;
var cursor = datastore.getDatabase().getCollection(coll).find(rewritten);
```

Option B: Apply ontology constraints to Morphia Filter/Query via ids.

- When the repo uses Morphia's typed query API instead of BSON, pre-compute the id set and constrain by _id:

```
Set<String> ids = edgeDao.srcIdsByDst(tenantId, predicate, dst);
if (ids.isEmpty()) {
  return List.of(); // short-circuit
}
query.filter(Filters.in("_id", ids));
```

Option C: Centralize in a tiny helper for developer ergonomics.

- Provide one helper in your application layer, invoked wherever policies inject additional constraints:

```
public final class OntologyFilterHelper {
  private final ListQueryRewriter rewriter;
```

```
  public OntologyFilterHelper(ListQueryRewriter r) { this.rewriter = r; }

  public Bson ensureHasEdge(Bson base, String tenantId, String predicate, String dst)
{
    return rewriter.rewriteForHasEdge(base, tenantId, predicate, dst);
  }
}
```

3.5. Where to hook materialization

- On entity changes that are sources or intermediates for chains:
- Order (placedBy, orderHasShipment), Customer (memberOf), Shipment (shipsTo), Address (locatedIn), Organization (ancestorOf/parent), and any Fulfillment/Returns entities.
- Recommended patterns:
- On-save/on-update hooks in your service layer call OntologyMaterializer.apply(…) with the explicit edges known from the entity snapshot.
- For intermediates (e.g., Address.region changed), enqueue affected sources for recomputation; use provenance to locate impacted edges.
- Provide nightly/backfill jobs for recomputing edges across a tenant when ontology rules evolve.

3.6. Security and multi-tenant considerations

- Edge rows include tenantId and should be validated/filtered by tenant on every operation.
- Never trust a client-supplied predicate or destination id blindly; combine with rule evaluation and whitelist allowed predicates per domain if needed.
- For shared resources across tenants (rare), model cross-tenant permissions at the policy layer; don’t reuse edges across tenants unless explicitly designed.

3.7. Developer workflow and DX checklist

- When writing a rule: use hasEdge("<predicate>", <rhs>) and rely on RuleContext variables for the destination when possible.
- When writing a list endpoint: read optional ontology filter hints from the policy layer; if present, apply ensureHasEdge(…) before find().
- When changing domain relationships: update predicates/chains and re-materialize; list/policy code stays unchanged.
- When indexing a new tenant: include the edges indexes early and validate via a smoke test query using ListQueryRewriter.

3.8. Cookbook: end-to-end example with Orders + Org

- Policy: allow LIST Order when hasEdge("placedInOrg", principal.orgRefName)

- Request lifecycle: 1) Security filter builds SecurityContext and RuleContext with tenantId and principal. 2) Policy evaluation returns a directive to constrain by hasEdge("placedInOrg", orgRefName). 3) Repo builds base filter (state != ARCHIVED, etc.). 4) Repo calls OntologyFilterHelper.ensureHasEdge(base, tenantId, "placedInOrg", orgRefName). 5) Mongo executes a single-hop query using materialized edges; results respect both policy and multi-tenancy.

3.9. Migration notes for teams using @Reference

- Keep existing references for write-side integrity and local joins where simple.
- Introduce ontology edges on hot read paths first; update policy rules to hasEdge and verify results.
- Gradually replace deep $lookup traversals with hasEdge-based rewrites.
- Ensure materialization hooks are deployed before removing data fields used as inputs to the ontology.