Ontologies in Quantum

Modeling Relationships That Are Resilient and Fast

Version 1.2.2-SNAPSHOT, 2025-11-02T21:32:23Z

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).

In Quarkus, all ontology components are CDI-managed. Inject EdgeDao and services via @Inject; indexes are ensured automatically at startup by OntologyMongoProducers. Configure collection/database with properties ontology.mongo.database and ontology.mongo.collection.edges.

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

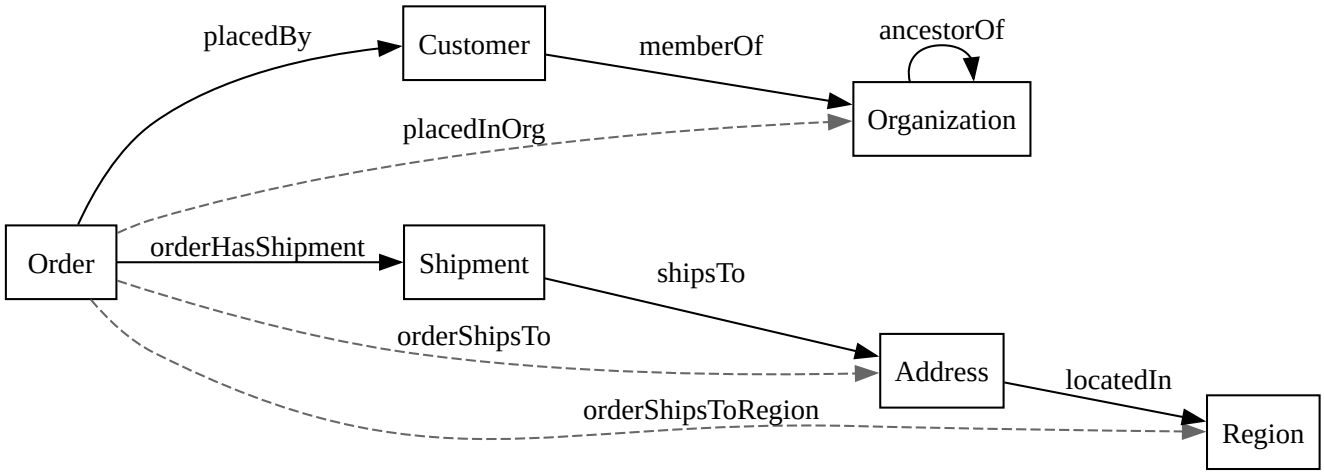
Key predicates (relationships)

- 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.

Diagram of the core classes and predicates for this example:



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

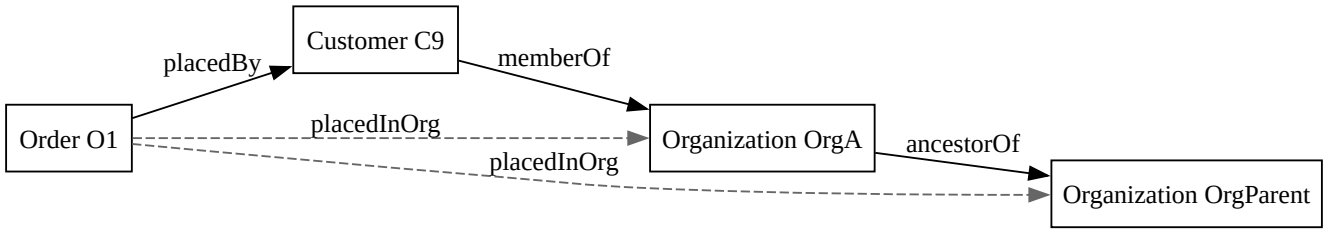
- Ensure EdgeDao has indexes on (tenantId, p, dst) and (tenantId, src, p).
- Use OntologyMaterializer when Order, Shipment, Customer, Address, or org hierarchy changes to keep edges fresh.
- Keep provenance in edge.prov (rule, inputs) so you can recompute or retract edges when source data changes.

Chapter 3. Visualizing ontology edges and provenance

This section uses simple diagrams to illustrate how explicit facts (edges) combine with ontology rules (property chains, inverses, transitivity) to produce inferred edges, and how provenance explains each inference.

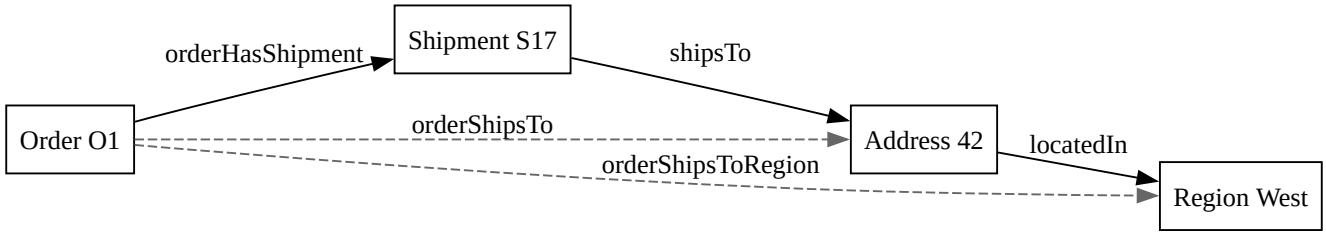
3.1. Example graph: Orders, Customers, Organizations

We start from the running example introduced above. The following diagram shows explicit edges in solid lines and inferred edges as dotted lines.



You can add more nodes and predicates (shipments, addresses, regions) and draw similar diagrams. The important idea is that you query for semantics (placedInOrg), while the reasoner ensures the materialized edges exist according to the rules.

3.2. Example graph: Shipments and Regions



3.3. What is provenance and how to read it

Every inferred edge includes a provenance payload prov that answers “why does this edge exist?”. It captures the rule used and the contributing input edges. With recent enrichments, inputs also include type information when available.

Here is an example edge document (simplified JSON for clarity) for the inference O1 --placedInOrg -→ OrgA via the chain placedBy □ memberOf ⇒ placedInOrg:

```
{
  "tenantId": "t1",
  "src": "O1",
  "srcType": "Order",
  "p": "placedInOrg",
  "dst": "OrgA",
```

```
  "dstType": "Organization",
  "inferred": true,
  "prov": {
    "rule": "chain",
    "inputs": {
      "chain": ["placedBy", "memberOf"],
      "inputs": [
        { "src": "O1",  "srcType": "Order",     "p": "placedBy", "dst": "C9",
"dstType": "Customer" },
        { "src": "C9",  "srcType": "Customer",  "p": "memberOf", "dst": "OrgA",
"dstType": "Organization" }
      ]
    }
  },
  "ts": "2025-10-20T10:01:02Z"
}
```

Notes: - rule indicates which inference rule created the edge (chain, inverse, symmetric, transitive, subPropertyOf). - inputs lists the contributing explicit edges (or minimal hops) that justified the inference. When types are known from domain/range or snapshots, srcType and dstType are included. - This provenance enables audit, troubleshooting, and selective recomputation when inputs change.

For a transitive example, an O1 --placedInOrg-→ OrgParent edge could have:

```
"prov": {
  "rule": "transitive",
  "inputs": {
    "prop": "placedInOrg",
    "inputs": [ { "src": "O1", "p": "placedInOrg", "dst": "OrgA", "srcType": "Order",
"dstType": "Organization" } ]
  }
}
```

Provenance is for explanation and maintenance. The runtime edges you query on remain simple (src, p, dst) with optional type fields for validation and indexing.

Chapter 4. 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.

4.1. 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.

4.2. 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.hasEdge(...), which converts hasEdge into a Morphia Filter limiting _id to the allowed source ids; compose it with your base filter via Filters.and(...).

4.3. 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)

4.4. 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);
  }
}
```

4.5. Where ontology materialization happens

- Automatic on write via repository hook:
- When you save or update an entity, the framework runs an internal PostPersistHook (OntologyWriteHook) that:
- extracts explicit edges from fields/getters annotated with @OntologyProperty using a startup-built metadata cache (no per-call reflection), and
- calls OntologyMaterializer to infer inverse, transitive, symmetric, and super-property edges and upsert them to the edges collection.
- This is enabled by default. To disable temporarily (e.g., for bulk loads), set: ontology.auto-materialize=false

4.5.1. Extending materialization with OntologyEdgeProvider (SPI)

The write hook is annotation-first and SPI-second:

- First, it extracts explicit edges from @OntologyProperty fields/getters on the entity being saved.
- Then, it discovers any CDI beans implementing com.e2eq.ontology.spi.OntologyEdgeProvider that support the entity type and merges their edges.
- Finally, it persists explicit edges, prunes stale ones, and rematerializes inferred edges (inverse, subPropertyOf, transitive, symmetric).

This keeps OntologyWriteHook thin and generic while allowing domain modules to contribute additional explicit edges without changing the core.

Basic contract

- Implementations must declare:
- boolean supports(Class<?> entityType) — return true for the entity types you handle.
- List<Reasoner.Edge> edges(String realmId, Object entity) — return explicit edges to add for that instance.
- Register the implementation as a CDI bean (e.g., @ApplicationScoped) in your module; Quarkus will discover it.
- Edges you return should use your domain refName (or id) for src/dst ids and the ontology type ids for srcType/dstType.

Example provider

```
@jakarta.enterprise.context.ApplicationScoped
public class UserCredentialEdgeProvider implements com.e2eq.ontology.spi
.OntologyEdgeProvider {
  @Override
  public boolean supports(Class<?> entityType) {
    return com.e2eq.framework.model.security.User.class.isAssignableFrom(entityType);
  }

  @Override
  public java.util.List<com.e2eq.ontology.core.Reasoner.Edge> edges(String realmId,
Object entity) {
    var user = (com.e2eq.framework.model.security.User) entity;
    String srcId = user.getRefName();
    if (srcId == null || srcId.isBlank()) return java.util.List.of();
    String credRef = user.getCredentialRefName();
    if (credRef == null || credRef.isBlank()) return java.util.List.of();
    return java.util.List.of(
      new com.e2eq.ontology.core.Reasoner.Edge(
        srcId, "User", "userHasCredential", credRef, "CredentialUserIdPassword",
false, java.util.Optional.empty()
      )
    );
  }
}
```

Merging behavior and lifecycle

- The hook calls all matching providers and concatenates their edges with the annotation-derived ones.
- It computes the diff against prior explicit edges for the same source and prunes any stale explicit edges per predicate.
- Then it invokes the reasoner/materializer to compute and upsert inferred edges.
- Finally, it executes cascade policies (e.g., ORPHAN_REMOVE/DELETE) declared on @OntologyProperty, using the prior vs. new explicit state.

ID and type resolution tips

- srcId: By default, the hook resolves an entity id using getRefName(), getId(), or toString() fallback. Prefer setting refName on your entities.
- dstId: You may point to another entity by refName string (no fetch required) if your domain guarantees uniqueness per type and tenant.
- srcType/dstType: Use the @OntologyClass(id=...) values for the types (e.g., "User", "CredentialUserIdPassword").
- realmId: The framework passes the current tenant/realm id; include it when persisting or resolving external references if needed.

Testing your provider

- You can write a Quarkus test that persists a source and any targets, calls OntologyWriteHook.afterPersist(tenant, entity), and asserts that:
- the annotation-derived edges exist, and
- your provider-derived edges are also present and marked as explicit (inferred=false).

Migration guidance

- Start annotation-first. Only add a provider when edges depend on computed fields, cross-aggregate ids, or non-entity sources.
- Keep providers small and focused per entity type. Avoid duplicating edges that annotations already cover.
- If multiple providers contribute the same edge, the repo upsert is idempotent.
- Intermediates and re-materialization:
- If you change entities that are not the source but affect chains (e.g., Address.region, Customer.memberOf), those sources will get updated the next time they are saved. For immediate consistency across a tenant, run the re-materialization job (see below).
- Provide nightly/backfill jobs for recomputing edges across a tenant when ontology rules evolve.

4.5.2. Configuration

```
# default is true
ontology.auto-materialize=true
```

4.5.3. Example: annotate and save — edges are materialized automatically

```
@OntologyClass(id = "Order")
public class Order {
  private String refName;

  @OntologyProperty(id = "placedBy", functional = true)
  public Customer getCustomer() { return customer; }
}

@OntologyClass(id = "Customer")
public class Customer {
  private String refName;

  @OntologyProperty(id = "memberOf")
  public Organization getOrg() { return org; }
}
```

- Saving an Order will automatically create explicit placedBy and, via chains/inference,

placedInOrg and inverse edges as defined in your ontology registry. No manual calls to OntologyMaterializer.apply(…) are required.

4.6. 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.

4.7. 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.

4.8. 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.

4.9. 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.

Chapter 5. Primer: First-time guide to core relationship semantics

If you are new to ontology terms, the following quick explanations will help you build intuition. Each concept explains what it means, a tiny example, and how it affects queries using hasEdge.

- Functional property
- What it means: For a given source entity, there is at most one target for this property. Think “single-valued.”
- Example: placedBy(Order, Customer) is functional because an Order is placed by exactly one Customer.
- Why it matters: On writes, the latest value replaces any previous one. In storage, you can enforce uniqueness for (tenantId, src, p). Queries are simpler because there’s at most one matching edge per source.
- In practice: When you save an Order with a different placedBy, the old edge is replaced so hasEdge("placedBy", customerX) reflects the latest truth.
- Transitive property
- What it means: The relationship “chains through” intermediates. If A relates-to B and B relates-to C with the same property p, then A relates-to C by p as well.
- Example: ancestorOf(Org, Org) is transitive: OrgA ancestorOf OrgB and OrgB ancestorOf OrgC implies OrgA ancestorOf OrgC.
- Why it matters: You can answer reachability questions without specifying every hop. The system can materialize or compute the closure so queries like hasEdge("ancestorOf", OrgC) return OrgA even if the path is multiple hops.
- In practice: Use transitive for hierarchies like parent/ancestor, partOf, locatedIn, reportsTo.
- subPropertyOf (property hierarchy)
- What it means: One property is a more specific form of another. If p □ q (“p is a sub-property of q”), then every (s, p, o) also counts as (s, q, o).
- Example: placedInOrg □ inOrg. If an Order is placedInOrg ACME, then it is also inOrg ACME.
- Why it matters: Queries written against the broader property (q) automatically include results from all its specializations (p). The system may materialize the super-property edges for performance.
- In practice: You can write policies/queries once for inOrg and still match edges stored as placedInOrg.
- inverseOf (inverse properties)
- What it means: Two properties point in opposite directions. If p is the inverse of q, then (s, p, o) implies (o, q, s).
- Example: parentOf □ childOf. If OrgA parentOf OrgB, then OrgB childOf OrgA.
- Why it matters: You only need to assert one direction; the system can infer the other. Queries

can use whichever direction is natural.

- In practice: If you stored parentOf, you can still query with hasEdge("childOf", OrgA) and find OrgB once inverses are materialized.

Putting it together - These traits combine. For example, placedInOrg can be declared a subPropertyOf inOrg; ancestorOf can be transitive; parentOf and childOf can be inverses. The reasoner uses these facts to materialize convenient edges so your queries remain simple and fast. - Your application code typically continues to use the same three query helpers: hasEdge, hasEdgeAny, notHasEdge. The richer semantics affect which edges exist, not how you call them.

Chapter 6. New ontology features in this release

This release expands the ontology engine and query integration to support a richer, OWL-RL–inspired subset while keeping the developer experience simple and backward compatible. The following capabilities are now available and used transparently by the materializer and query rewriter:

6.1. Feature overview

- subClassOf (class hierarchy)
- Model taxonomies (e.g., Order Entity, Employee Person). Used for validation and optional type filters.
- subPropertyOf (property hierarchy)
- If p q and you assert (s, p, o), the system materializes (s, q, o). Queries on q will match p's edges as well.
- inverse properties
- If p is the inverse of q, asserting (s, p, o) will materialize (o, q, s). Example: parentOf childOf.
- symmetric properties
- For symmetric p, asserting (s, p, o) materializes (o, p, s). Example: peerOf between organizations.
- transitive properties
- For transitive p, if (s, p, m) and (m, p, o) then (s, p, o) is inferred. Typical for ancestorOf, partOf, locatedIn, memberOf within hierarchical contexts.
- functional properties
- For functional p, each source s has at most one destination o. Upserts replace the prior value. Useful for single-valued relations like placedBy on Order.

All inferred edges carry inferred=true and provenance containing the rule used (e.g., transitive, inverse, subPropertyOf). These edges live in the same collection and are fully queryable.

6.2. Business problems solved

- E-commerce organizational access
- Problem: "Show me all orders placed in ACME (including subsidiaries)."
- Solution: Define placedBy, memberOf, placedInOrg with chain placedBy memberOf ⇒ placedInOrg, and make ancestorOf transitive. Either materialize closure with a chain placedInOrg ancestorOf ⇒ placedInOrg or use p being transitive. Queries use hasEdge("placedInOrg", orgId).
- HR management

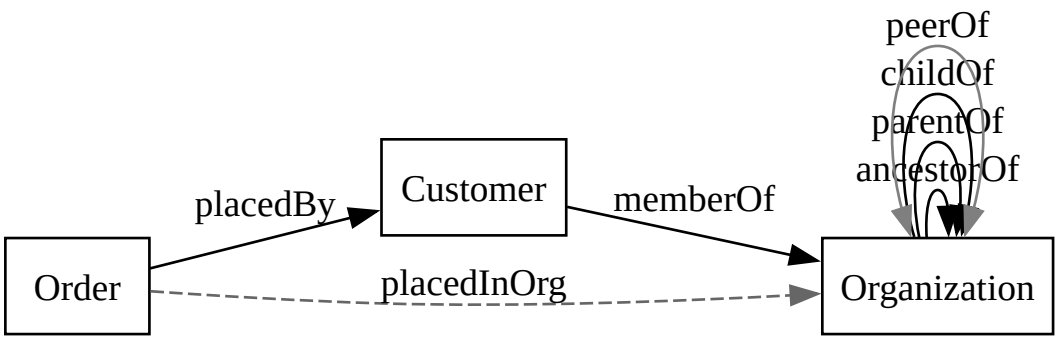
- Problem: “List employees who report to VP_X directly or indirectly.”
- Solution: Define reportsTo as transitive. Query hasEdge("reportsTo", "VP_X").
- Third-party and data lineage
- Problem: “Which datasets are derived from Source S?”
- Solution: Define derivedFrom as transitive and inverse derivedInto. Record symmetric peerOf for bidirectional partnerships where relevant.
- Supplier networks (supply chain)
- Problem: “Find parts supplied by a vendor’s parent company’s peers.”
- Solution: Use subPropertyOf to roll up specialized edges into a common supplies predicate, define parentOf/childOf inverses, and peerOf symmetric at the org level.

6.3. How to code it with current APIs

Below are concise examples showing how to define the ontology, materialize inferences, and query with existing components. These mirror the types already present in this repository and require no grammar changes.

6.3.1. 1) Define ontology (programmatic TBox)

A small TBox-level view of the classes and predicates we define:



Java (at startup, e.g., in a CDI producer):

```
@Produces @Singleton
public OntologyRegistry ontologyRegistry() {
  Map<String, OntologyRegistry.ClassDef> classes = Map.of(
      "Order",       new OntologyRegistry.ClassDef("Order", Set.of(), Set.of(), Set.
of()),
      "Customer",    new OntologyRegistry.ClassDef("Customer", Set.of(), Set.of(),
Set.of()),
      "Organization",new OntologyRegistry.ClassDef("Organization", Set.of(), Set.of(),
Set.of())
  );
```

```
  Map<String, OntologyRegistry.PropertyDef> props = new HashMap<>();
  // Single-valued ⇒ functional
  props.put("placedBy",   new OntologyRegistry.PropertyDef(
      "placedBy", Optional.of("Order"), Optional.of("Customer"), false, Optional.
empty(),
      false, /*transitive*/ false, /*symmetric*/ true /*functional*/, Set.of()
  ));
  props.put("memberOf",   new OntologyRegistry.PropertyDef(
      "memberOf", Optional.of("Customer"), Optional.of("Organization"), false,
Optional.empty(),
      false, false, false, Set.of()
  ));
  props.put("placedInOrg",new OntologyRegistry.PropertyDef(
      "placedInOrg", Optional.of("Order"), Optional.of("Organization"), false,
Optional.empty(),
      false, false, false, Set.of("inOrg") // subPropertyOf example
  ));
  props.put("inOrg",      new OntologyRegistry.PropertyDef(
      "inOrg", Optional.of("Order"), Optional.of("Organization"), false, Optional
.empty(),
      false, false, false, Set.of()
  ));
  // Transitive ancestor relation
  props.put("ancestorOf", new OntologyRegistry.PropertyDef(
      "ancestorOf", Optional.of("Organization"), Optional.of("Organization"), false,
Optional.empty(),
      true, /*transitive*/ false, false, Set.of()
  ));
  // Inverses and symmetric
  props.put("parentOf",   new OntologyRegistry.PropertyDef(
      "parentOf", Optional.of("Organization"), Optional.of("Organization"), false,
Optional.of("childOf"),
      false, false, false, Set.of()
  ));
  props.put("childOf",    new OntologyRegistry.PropertyDef(
      "childOf", Optional.of("Organization"), Optional.of("Organization"), true,
Optional.empty(),
      false, false, false, Set.of()
  ));
  props.put("peerOf",     new OntologyRegistry.PropertyDef(
      "peerOf", Optional.of("Organization"), Optional.of("Organization"), false,
Optional.empty(),
      false, /*transitive*/ true, /*symmetric*/ false, /*functional*/ Set.of()
  ));

  List<OntologyRegistry.PropertyChainDef> chains = List.of(
      // Order --placedBy--> Customer --memberOf--> Org  ⇒ Order --placedInOrg--> Org
      new OntologyRegistry.PropertyChainDef(List.of("placedBy", "memberOf"),
"placedInOrg"),
      // Include ancestor closure for placedInOrg results
```

```
      new OntologyRegistry.PropertyChainDef(List.of("placedInOrg", "ancestorOf"),
"placedInOrg")
  );

  return OntologyRegistry.inMemory(new OntologyRegistry.TBox(classes, props, chains));
}
```

Notes: - subPropertyOf is modeled via the PropertyDef.subPropertyOf set. - Inverse and symmetric are encoded in PropertyDef.inverseOf and PropertyDef.symmetric. - Transitivity is toggled via PropertyDef.transitive.

6.3.2. 2) Materialize inferences when data changes

Use OntologyMaterializer or call the Reasoner directly and upsert edges via OntologyEdgeRepo.

```
@Inject ForwardChainingReasoner reasoner;
@Inject OntologyRegistry ontologyRegistry;
@Inject OntologyEdgeRepo edgeRepo;

public void onOrderSaved(String tenant, String orderId, String customerId, String
orgId) {
  List<Reasoner.Edge> explicit = List.of(
      new Reasoner.Edge(orderId, "Order", "placedBy", customerId, "Customer", false,
Optional.empty()),
      new Reasoner.Edge(customerId, "Customer", "memberOf", orgId, "Organization",
false, Optional.empty())
  );
  Reasoner.EntitySnapshot snap = new Reasoner.EntitySnapshot(tenant, orderId, "Order",
explicit);
  Reasoner.InferenceResult out = reasoner.infer(snap, ontologyRegistry);
  for (Reasoner.Edge e : out.addEdges()) {
    Map<String,Object> prov = e.prov().map(p -> Map.<String,Object>of("rule", p.rule(
), "inputs", p.inputs())).orElse(Map.of());
    edgeRepo.upsert(tenant, e.srcType(), e.srcId(), e.p(), e.dstType(), e.dstId(),
true, prov);
  }
}
```

The materializer computes and persists: - subPropertyOf roll-ups (e.g., placedInOrg □ inOrg ⇒ write inOrg) - inverse/symmetric counterparts - transitive closures (bounded to the snapshot inputs; you can also chain for closures)

6.3.3. 3) Query with ListQueryRewriter (Quarkus/Morphia)

Use hasEdge/hasEdgeAny/notHasEdge to turn ontology constraints into Morphia filters over your entity collection. The rewriter internally fetches the set of source ids from the edge store and builds an in("refName", ...) filter that composes with your attribute predicates.

```
@Inject ListQueryRewriter queryRewriter;
@Inject MorphiaDatastore datastore;

public List<TestOrder> openOrdersIn(String tenant, String orgId) {
  var status = dev.morphia.query.filters.Filters.eq("status", "OPEN");
  var inOrg = queryRewriter.hasEdge(tenant, "placedInOrg", orgId);
  var combined = dev.morphia.query.filters.Filters.and(status, inOrg);
  return datastore.find(TestOrder.class).filter(combined).iterator().toList();
}
```

To include multiple orgs:

```
var f = queryRewriter.hasEdgeAny(tenant, "placedInOrg", List.of("ORG-ACME", "ORG-
GLOBEX"));
```

To exclude restricted orgs:

```
var f = queryRewriter.notHasEdge(tenant, "placedInOrg", "ORG-RESTRICTED");
```

6.3.4. Functional properties at write time

For single-valued relationships declared functional, configure a unique index over (tenantId, src, p) in your edge store (or rely on repo semantics) and treat upserts as replacements.

Practical guidance: - Mark naturally single-valued links functional (placedBy, primaryOwner). - On write, remove or overwrite any prior dst for the same (tenant, src, p).

6.4. Backward compatibility and migration

- Grammar/API: No changes needed. The same hasEdge/hasEdgeAny/notHasEdge functions work; results simply become more expressive as ontology features are enabled.
- Feature flags: You may enable features per property (e.g., transitive) gradually. Queries remain stable.
- Provenance: Keep prov so you can re-compute or retract inferred edges when inputs or rules change.

6.5. Troubleshooting and tips

- Empty result from hasEdge: Means no matching edges exist; check whether inferences are being materialized for the predicate you expect.
- Tenant scoping: All repo calls include tenantId; ensure you pass the correct tenant in both write and read paths.
- Indexes: For heavy reads, ensure compound indexes on (tenantId, p, dst) and (tenantId, src, p).

For functional, consider a unique (tenantId, src, p).

- Testing: Use Quarkus @QuarkusTest to wire CDI and a test OntologyRegistry producer. Populate explicit edges, run the reasoner, then assert with the repo and query rewriter as shown in the integration tests included with the repository.

Chapter 7. Model-first ontology with annotations and MorphiaOntologyLoader

This section shows how to use your Morphia model classes as the source of truth for the ontology. You annotate model classes/fields to declare ontology concepts and property traits; at startup, MorphiaOntologyLoader scans MorphiaDatastore.getMapper() to build the OntologyRegistry automatically.

7.1. Why model-first?

- Single source of truth: semantics live next to the data model; less drift between code and config.
- Safer refactors: renames/types evolve in code and the ontology follows.
- No separate YAML/JSON needed for most use cases; optional overrides remain possible later if needed.

7.2. Annotations overview

Two lightweight annotations are used on model classes and fields. They are provided by quantum-ontology-core and can be added to any Morphia-mapped entity (deriving from UnversionedBaseModel).

- @OntologyClass
- id: optional explicit class id (defaults to the simple class name)
- subClassOf: optional extra parents (in addition to Java inheritance)
- @OntologyProperty
- id: optional explicit property id (defaults to the field name)
- subPropertyOf: property hierarchy roll-up targets
- inverseOf: name of the inverse property (declare on one side)
- transitive: whether the property is transitive
- symmetric: whether the property is symmetric
- functional: at most one target per source (defaults to true for single-valued fields)
- domain: override inferred domain (defaults to declaring @OntologyClass)
- range: override inferred range (defaults to the field/element type)
- ref: target ontology class id when this field represents a relationship to another ontology type
- relation: the multiplicity/cardinality for the relationship (NONE, ONE_TO_ONE, ONE_TO_MANY, MANY_TO_ONE, MANY_TO_MANY)
- edgeType: the edge label used in the ontology graph; if omitted, falls back to id or field name
- inverseOfEdge: optional inverse edge label if you prefer naming inverses at the edge level
- materializeEdge: when false, keeps the reference value but skips edge creation (defaults to true)

- cascade: optional cascade policies (NONE by default). Supports ORPHAN_REMOVE, DELETE, and BLOCK_IF_REFERENCED; see below.
- cascadeDepth: limit for recursive cascades (default 1); used by DELETE cascades to bound depth and prevent long chains

functional is inferred true for single-valued fields and false for collections/maps unless you override it.

7.2.1. Relationship semantics on @OntologyProperty

When a field is annotated with @OntologyProperty and declares any of ref, relation, or edgeType, the framework treats it as a relationship and will materialize an ontology edge for it on persist/update.

Key attributes: - ref: Names the target ontology class id. If omitted, the loader attempts to infer it from the Java type of the field or its generic element type. - relation: Declares multiplicity. If not set, it is inferred from the field type: collections imply plural (ONE_TO_MANY), scalars imply singular (MANY_TO_ONE) by convention. - edgeType: Controls the predicate id used to write/read edges. If absent, the property id (or field name) is used. - materializeEdge: Set to false to store the reference value without creating graph edges; traversal and inferences will ignore this property.

Examples:

```
@OntologyClass(id = "Order")
@Entity("orders")
public class Order extends UnversionedBaseModel {
  // Many orders reference one Customer (MANY_TO_ONE); name the edge explicitly
  @OntologyProperty(
    id = "customer",
    ref = "Customer",
    relation = RelationType.MANY_TO_ONE,
    edgeType = "CUSTOMER_OF"
  )
  private String customerId; // or Customer if you store the object
}

@OntologyClass(id = "SCorp")
@Entity("scorps")
public class SCorp extends UnversionedBaseModel {
  // One SCorp authorizes many Resolutions
  @OntologyProperty(
    id = "resolutionIds",
    ref = "Resolution",
    relation = RelationType.ONE_TO_MANY,
    edgeType = "AUTHORIZES_RESOLUTION"
  )
  private List<String> resolutionIds;
}
```

Runtime materialization (conceptual): - (Order) -[CUSTOMER_OF]→ (Customer) - (SCorp) -[AUTHORIZES_RESOLUTION]→ (Resolution)

If ref is provided but edgeType is omitted, the system uses the property id (or field name). Consider using descriptive edge names for durability of queries.

7.2.2. Cascading policies for relationships

Cascading is opt-in and conservative. It controls what happens to related targets and edges when a relationship is modified or a source is deleted.

Supported policies today: - ORPHAN_REMOVE: On update, when an element is removed from a collection (or a singular reference is replaced), the framework deletes the removed target if and only if no other sources still reference it via the same predicate. This is useful for owned child aggregates. - UNLINK: Edges are always pruned to reflect the current snapshot; you can think of this as implicit and always-on for edge hygiene.

Delete-time cascades (also supported): - DELETE: When deleting the source, also delete targets up to cascadeDepth with cycle guards. Targets are deleted only for predicates that declare DELETE on the source side. - BLOCK_IF_REFERENCED: Before deleting a target (either directly or via DELETE cascade), if the target is still referenced by another source via the same predicate, the deletion is skipped (or blocked) to preserve referential integrity.

Example (delete children when parent is deleted, with safety guards):

```
@OntologyClass(id = "Parent")
@Entity("parents")
public class Parent extends UnversionedBaseModel {
  @OntologyProperty(
    id = "children",
    edgeType = "HAS_CHILD",
    ref = "Child",
    relation = RelationType.ONE_TO_MANY,
    cascade = { CascadeType.DELETE, CascadeType.BLOCK_IF_REFERENCED },
    cascadeDepth = 1 // delete only direct children
  )
  private List<Child> children = new ArrayList<>();
}
```

Behavior: - Deleting a Parent removes explicit edges and, with DELETE, deletes its Child documents up to the specified depth. - With BLOCK_IF_REFERENCED, a Child that is still referenced by another Parent is not deleted; only the edge from the deleted Parent is removed. - Cycles are guarded by a visited set; depth prevents long delete chains.

See integration tests in the ontology-mongo module (CascadeDeleteIT) for expected behavior.

Example (owned children with orphan removal):

```
@OntologyClass(id = "Parent")
@Entity("parents")
public class Parent extends UnversionedBaseModel {
  @OntologyProperty(
    id = "children",
    edgeType = "HAS_CHILD",
    ref = "Child",
    relation = RelationType.ONE_TO_MANY,
    cascade = { CascadeType.ORPHAN_REMOVE }
  )
  private List<Child> children = new ArrayList<>();
}
```

Behavior: - Persisting a Parent materializes edges (P --HAS_CHILD-→ C) for current children. - Removing a Child from the collection prunes the edge. If no other Parent points to that Child via HAS_CHILD, the Child entity is deleted. - If the Child is still referenced by another Parent, it is retained.

Verification: See integration tests under quantum-ontology-mongo it module (CascadeOrphanRemoveIT) covering both deletion and shared reference cases.

7.2.3. Business benefits of ontology relationships and cascade

- Faster queries and policies: Pre-materialized edges make filtering by relationships a single indexed lookup instead of deep joins or application-side traversals.
- Safer evolution: Edge names and ontology traits remain stable as you refactor field names/types; queries continue to work.
- Cleaner models: Relationship intent lives in one place (@OntologyProperty) rather than being scattered across repos and services.
- Reduced data drift: Automated edge pruning and orphan removal keep the graph consistent with your source-of-truth objects.
- Auditability: Provenance on inferred edges documents why links exist, aiding debugging and compliance.

7.3. Example: Orders placed in an Organization (e-commerce)

Below we implement the same business example covered earlier using model annotations.

```
@OntologyClass(id = "Order")
@Entity(value = "orders")
public class Order extends UnversionedBaseModel {
  private String status;

  // Order --placedBy--> Customer (single-valued ⇒ functional)
```

```
  @OntologyProperty(id = "placedBy", inverseOf = "placed", functional = true)
  private Customer placedBy;
}

@OntologyClass(id = "Customer")
@Entity(value = "customers")
public class Customer extends UnversionedBaseModel {
  // Customer --memberOf--> Org (transitive across org tree)
  @OntologyProperty(id = "memberOf", transitive = true)
  private Org memberOf;
}

@OntologyClass(id = "Organization")
@Entity(value = "orgs")
public class Org extends UnversionedBaseModel {
  // Org --parentOf--> Org, with inverse childOf
  @OntologyProperty(id = "parentOf", inverseOf = "childOf")
  private Org parent;

  // Optional symmetric relationship between peers
  @OntologyProperty(id = "peerOf", symmetric = true)
  private Set<Org> peers;
}
```

Optionally, model a super-property to roll up specialized edges:

```
@OntologyClass(id = "Order")
public class Order extends UnversionedBaseModel {
  @OntologyProperty(id = "placedBy", inverseOf = "placed", functional = true)
  private Customer placedBy;

  // Roll-up predicate for org membership of orders; placedInOrg ⊑ inOrg
  @OntologyProperty(id = "placedInOrg", subPropertyOf = {"inOrg"})
  private Org inOrgDirect;
}
```

With these annotations in place, the loader will infer: - Classes: Order, Customer, Organization (+ hierarchy from Java inheritance) - Properties: placedBy (functional), memberOf (transitive), parentOf ⁻¹ childOf (inverse), peerOf (symmetric) - subPropertyOf: placedInOrg ⊑ inOrg (if declared)

7.4. How the registry is produced (Quarkus CDI)

OntologyCoreProducers wires MorphiaOntologyLoader by default. On startup it scans your models and produces a singleton OntologyRegistry.

```
@ApplicationScoped
public class OntologyCoreProducers {
  @Inject MorphiaDatastore morphiaDatastore;
```

```
  @Produces @Singleton
  public OntologyRegistry ontologyRegistry() {
    try {
      MorphiaOntologyLoader loader = new MorphiaOntologyLoader(morphiaDatastore);
      OntologyRegistry reg = loader.load();
      return isEmpty(reg) ? emptyRegistry() : reg;
    } catch (Throwable t) {
      return emptyRegistry();
    }
  }
}
```

If you need manual control (tests, migrations), you can call the loader directly:

```
MorphiaOntologyLoader loader = new MorphiaOntologyLoader(datastore);
OntologyRegistry registry = loader.load();
```

Keep annotations minimal. Only add traits that aren't obvious from the Java type system (e.g., transitive, symmetric, inverse, subPropertyOf).

7.5. Defining an ontology with YAML (YamlOntologyLoader)

You can also author your ontology in a simple YAML file and load it at runtime using YamlOntologyLoader. This is useful when: - You want product or ops teams to evolve predicates and chains without changing Java models. - You need to add an overlay (additional properties or chains) on top of what is discovered from annotations.

YAML structure supported by YamlOntologyLoader: - classes: list of concept ids - properties: list of predicate definitions with optional fields - id: predicate name - domain: optional class id for the source - range: optional class id for the target - inverseOf: optional inverse predicate id (declare on either side; loader wires both) - transitive: boolean - symmetric: boolean - relation: NONE | ONE_TO_ONE | ONE_TO_MANY | MANY_TO_ONE | MANY_TO_MANY (if provided, overrides functional) - functional: boolean (deprecated in favor of relation; still supported for backward compatibility) - subPropertyOf: list of super-predicate ids - chains: list of property chains - chain: [p1, p2, ...] - implies: r

Example YAML (full file available in docs at user-guide/examples/ontology.yaml):

```
# Example ontology.yaml used by YamlOntologyLoader
version: 1
classes:
  - id: Order
  - id: Customer
  - id: Organization
```

```yaml
properties:
  - id: placedBy
    domain: Order
    range: Customer
    inverseOf: placed
    functional: true
  - id: placed
    domain: Customer
    range: Order
    inverseOf: placedBy
  - id: memberOf
    domain: Customer
    range: Organization
  - id: placedInOrg
    domain: Order
    range: Organization
    subPropertyOf: [ "inOrg" ]
  - id: inOrg
    domain: Order
    range: Organization
  - id: parentOf
    domain: Organization
    range: Organization
    inverseOf: childOf
  - id: childOf
    domain: Organization
    range: Organization
    inverseOf: parentOf
  - id: ancestorOf
    domain: Organization
    range: Organization
    transitive: true
  - id: peerOf
    domain: Organization
    range: Organization
    symmetric: true
chains:
  - chain: [ "placedBy", "memberOf" ]
    implies: "placedInOrg"
  - chain: [ "placedInOrg", "ancestorOf" ]
    implies: "placedInOrg"
```

Loading YAML in code:

- Direct usage (plain Java):

```java
import com.e2eq.ontology.core.OntologyRegistry;
import com.e2eq.ontology.core.YamlOntologyLoader;

YamlOntologyLoader loader = new YamlOntologyLoader();
```

```
// From classpath resource (place ontology.yaml under src/main/resources)
OntologyRegistry.TBox tbox1 = loader.loadFromClasspath("/ontology.yaml");

// From a file system path
java.nio.file.Path path = java.nio.file.Path.of("config", "ontology.yaml");
OntologyRegistry.TBox tbox2 = loader.loadFromPath(path);

// Build a registry (in-memory) from a TBox
OntologyRegistry registry = new com.e2eq.ontology.core.InMemoryOntologyRegistry(tbox2
);
```

- Quarkus auto-wiring (preferred):

OntologyCoreProducers will try to overlay YAML on top of the Morphia-discovered TBox at startup. It looks for one of the following, in order: - A system property ontology.yaml.path pointing to the file. - An environment variable ONTOLOGY_YAML pointing to the file. - A conventional file config/ontology.yaml in the working directory. - A classpath resource /ontology.yaml.

If found, it loads the YAML TBox and merges it with the base TBox from annotations using OntologyMerger, then validates using OntologyValidator.

Keep overlays additive. If you need to override flags (e.g., mark an existing property as transitive), ensure your overlay defines the same predicate id with the desired traits; the merger will apply the union/override semantics where supported.

7.6. Materializing inferences from annotated models

Materialization is automatic on save/update through the repository hook; you do not need to assemble explicit edge lists or call OntologyMaterializer manually.

- What happens:
- The framework inspects fields/getters annotated with @OntologyProperty and extracts explicit edges.
- The reasoner applies inverse, symmetric, transitive, and subPropertyOf traits from the registry and upserts inferred edges.
- When to trigger re-materialization manually:
- After ontology rule changes (e.g., you mark a property transitive or add an inverse), run your re-materialization job to recompute edges for existing entities in a tenant.
- During bulk imports, you can disable auto-materialize and run the job afterward.

Example: disable auto-materialize for a batch, then rebuild

```
ontology.auto-materialize=false
```

```
@Inject OntologyRebuilder rebuilder; // your maintenance utility

public void runBackfill(String tenantId) {
  rebuilder.recomputeTenant(tenantId);
}
```

7.7. Querying using ListQueryRewriter (unchanged)

Your query code does not change. It benefits from materialized edges that now carry richer semantics from annotations.

```
@Inject ListQueryRewriter queryRewriter;
@Inject MorphiaDatastore datastore;

public List<Order> openOrdersIn(String tenant, String orgId) {
  var status = dev.morphia.query.filters.Filters.eq("status", "OPEN");
  var inOrg = queryRewriter.hasEdge(tenant, "placedInOrg", orgId);
  var combined = dev.morphia.query.filters.Filters.and(status, inOrg);
  return datastore.find(Order.class).filter(combined).iterator().toList();
}
```

To include subsidiaries via ancestor/parent relationships, either: - Declare ancestorOf as transitive in the model and materialize a closure, or - Add a property chain placedInOrg □ ancestorOf ⇒ placedInOrg in your registry initialization.

7.8. Business case recipes with annotations

- E-commerce org access
- Annotate placedBy (functional), memberOf (transitive), and optionally parentOf (inverse childOf). Materialize placedInOrg via a property chain. Query hasEdge("placedInOrg", org).
- HR reporting lines
- Annotate reportsTo as transitive on Employee; query hasEdge("reportsTo", managerId).
- Supplier networks
- Annotate supplies as a base property; use subPropertyOf to roll up specialized suppliesDirect and suppliesViaSubsidiary to supplies; annotate peerOf on Org as symmetric.

7.9. Troubleshooting

- The registry is empty at runtime
- Ensure your models are discovered by Morphia (annotated with @Entity) and that the Quarkus application initializes MorphiaDatastore before the ontology producer.
- Inverse not applied

- Declare inverseOf on one side only; the loader will wire both directions.
- Functional not enforced
- For functional properties, ensure your edge repository enforces unique (tenantId, src, p) or that your write path overwrites prior values.

7.10. Migration strategy

- Start by adding @OntologyProperty to the most important relationships; don't attempt to annotate everything at once.
- Keep existing @Reference fields if you have them; use edges for query-time semantics and policy integration first.
- Add tests that verify the loader discovered your properties (including inverses/subPropertyOf) and that queries via hasEdge behave as expected.

7.11. Modeling guidance: annotate existing id/reference getters (keep API and DB clean)

- Prefer annotating getters that already exist in your schema (String ids or existing @Reference fields). Avoid adding embedded objects solely to drive ontology.
- If you do need a synthetic accessor purely for ontology, hide it from API docs and persistence using @JsonIgnore, @Schema(hidden = true), and @dev.morphia.annotations.Transient.

Recommended pattern: annotate an existing id getter (Swagger unchanged)

```
import com.e2eq.ontology.annotations.OntologyClass;
import com.e2eq.ontology.annotations.OntologyProperty;
import dev.morphia.annotations.Entity;

@Entity
@OntologyClass(id = "Order")
public class Order extends UnversionedBaseModel {
  // Existing field that is already part of your API/DB contract
  private String customerRefName;

  public String getCustomerRefName() { return customerRefName; }
  public void setCustomerRefName(String v) { this.customerRefName = v; }

  // Drives ontology edges without changing Swagger: the getter already exists and
returns a String id
  @OntologyProperty(id = "placedBy", functional = true)
  public String customerIdForEdges() { return getCustomerRefName(); }
}
```

Because AnnotatedEdgeExtractor treats CharSequence targets as ids, annotating a String-returning getter produces edges without introducing new embedded objects. Your Swagger/OpenAPI and

persisted shape remain unchanged.

Optional: use an existing @Reference instead of an id

```
@Entity
@OntologyClass(id = "Order")
public class Order extends UnversionedBaseModel {
  @dev.morphia.annotations.Reference
  private Customer customer; // already in your schema

  @OntologyProperty(id = "placedBy", inverseOf = "placed", functional = true)
  public Customer getCustomer() { return customer; }
}
```

If you must add an ontology-only accessor, hide it

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import io.swagger.v3.oas.annotations.media.Schema;
import dev.morphia.annotations.Transient;

@Entity
@OntologyClass(id = "Order")
public class Order extends UnversionedBaseModel {
  private String orgRefName; // real field

  // Hidden from API docs and persistence, used only to emit an ontology edge
  @JsonIgnore
  @Schema(hidden = true)
  @Transient
  @OntologyProperty(id = "placedInOrg", subPropertyOf = {"inOrg"})
  public String placedInOrgEdge() { return orgRefName; }
}
```

These patterns keep your REST and database contracts clear while allowing ontology materialization to happen automatically on writes.