Table of Contents

Chapter 1. Relationship hydration with expand(path)

This section documents the new relationship traversal and hydration capability added to the BIAPI query language. It preserves backward compatibility and introduces no SQL-like syntax.

1.1. How it works

- You add one or more expand(path) directives to your query string to request hydration of related entities at specific paths.
- The planner inspects the query. If any expand(...) directives are present, it selects AGGREGATION mode; otherwise, FILTER mode.
- In FILTER mode, the system converts the query to a Morphia Filter and queries a single MongoDB collection.
- In AGGREGATION mode, the system constructs an aggregation pipeline (with $lookup, $unwind, etc.) to materialize related documents. Execution may be guarded by a feature flag in the QueryGatewayResource.
- Projection can be applied at the root and per expansion using fields:[+/-] syntax. Unknown projection paths are hard errors.

See also:

- Planner and QueryGateway for mode selection and Java APIs
- QueryGateway REST endpoints to plan or execute queries over HTTP

1.2. What is expand(path)?

Use expand(path) to hydrate properties that are modeled as EntityReference or ontology-annotated relations. The path is dot-based and may include array wildcards [*] between segments.

- Single reference: expand(customer)
- Array of references: expand(items[*].product)
- Nested arrays: expand(patient.visits**[].diagnoses[**])

Expansion is a query-time operation: the related document(s) are materialized and embedded at the specified path in the result.

1.3. Filtering related entities (inline)

Continue to use your existing filter primitives in the same query string to filter the root collection. Filtering on related entities will be supported via either:

- Inline path blocks: path { key:value, key2:^[v1,v2] }

- Or filters inside expand(...): expand(customer, status:active)

Note: In v1, parsing of expand(path) is enabled; related-entity filters inside expand and inline path blocks will be introduced with the planner/compiler work.

1.4. Projection with fields:[+/-]

A unified projection syntax can be used both at the root and per expansion.

- Inclusion mode: fields:[+_id,+total,+customer.name]
- Exclusion mode: fields:[-internalNotes,-secret]
- Inside expand: expand(customer, fields:[+name,+tier,-ssn])

Rules: - If any + appears → include only those paths, then apply - as carve-outs. - If only - appears → include all by default and remove listed paths. - _id keeps current default behavior unless explicitly set.

1.5. Examples

- Hydrate a single reference and keep a few fields

```
q = "realm:acme && expand(customer, fields:[+name,+tier]) &&
fields:[+_id,+total,+customer.name]"
```

- Hydrate array of product references inside line items, filter to active products, project a couple of fields

```
q = "expand(items[*].product, active:true, fields:[+sku,+title]) &&
fields:[+_id,+items.product.sku]"
```

- Filter by a related product without projecting it (inline block form)

```
q = "items[*].product{ active:true, sku:^[A123,B456] } && fields:[+_id,+total]"
```

1.6. Semantics and limits

- Backward compatible: if no expand(...) or related-path filters are present, queries run as they do today (single-collection).
- Depth for nested paths is bounded; on mixed array/object hops, traversal stops at the first non-reference boundary.
- Unknown projection paths are hard errors (the request fails with a clear message).

See also: Planner and QueryGateway

1.7. Aggregation examples: what expand(...) compiles to

This section shows typical MongoDB aggregation stages the planner emits when expand(path) is present. Exact collection names and field names depend on your model mapping. If the collection cannot be resolved at plan-time, the planner will emit a placeholder from: "*unknown*"; the REST gateway returns HTTP 422 on execution in that case.

Tip: You can inspect the chosen mode and expand paths with POST /api/query/plan. See QueryGateway REST endpoints.

1.7.1. Single reference: expand(customer)

Query

```
q = "status:active && expand(customer, fields:[+name,+tier]) &&
fields:[+_id,+total,+customer]"
```

Typical pipeline skeleton

```
[
  { "$match": { "status": "active" } },
  { "$lookup": {
      "from": "customers",
      "localField": "customerId",
      "foreignField": "_id",
      "as": "__exp_customer"
  }},
  { "$unwind": { "path": "$__exp_customer", "preserveNullAndEmptyArrays": true }},
  { "$set": { "customer": "$__exp_customer" }},
  { "$project": { "_id": 1, "total": 1, "customer": { "name": 1, "tier": 1 } } }
]
```

Result shape (conceptual)

```
{
  "_id": "O123",
  "total": 99.50,
  "customer": { "_id": "C1", "name": "Acme", "tier": "GOLD" }
}
```

Notes - The intermediate field __exp_customer is an internal staging alias; the planner may omit it in future. - Per-expansion projection fields:[+name,+tier] narrows the embedded document via $project.

1.7.2. Array of references: expand(items[*].product)

Query

```
q = "expand(items[*].product, fields:[+sku,+title]) && fields:[+_id,+items.product]"
```

Typical pipeline patterns (two common strategies)

1) Unwind-and-regroup (easy to read)

```
[
  { "$unwind": { "path": "$items", "preserveNullAndEmptyArrays": true }},
  { "$lookup": {
      "from": "products",
      "localField": "items.productId",
      "foreignField": "_id",
      "as": "__exp_product"
  }},
  { "$unwind": { "path": "$__exp_product", "preserveNullAndEmptyArrays": true }},
  { "$set": { "items.product": "$__exp_product" }},
  { "$group": {
      "_id": "$_id",
      "doc": { "$first": "$$ROOT" },
      "items": { "$push": "$items" }
  }},
  { "$replaceRoot": { "newRoot": { "$mergeObjects": ["$doc", { "items": "$items" }] }
}},
  { "$project": { "_id": 1, "items.product": { "sku": 1, "title": 1 } } }
]
```

2) Map-with-lookup (keeps array order without grouping; requires $lookup with pipeline + $map)

```
[
  { "$set": {
      "items": {
        "$map": {
          "input": "$items",
          "as": "it",
          "in": {
            "$let": {
              "vars": {
                "p": { "$first": {
                  "$lookup": {
                    "from": "products",
                    "let": { "pid": "$$it.productId" },
                    "pipeline": [ { "$match": { "$expr": { "$eq": ["$_id", "$$pid"] }
} } ],
                    "as": "p"
```

```
                }
              }}
            },
            "in": { "$mergeObjects": [ "$$it", { "product": "$p" } ] }
          }
        }
      }
    }
  }},
  { "$project": { "_id": 1, "items.product": { "sku": 1, "title": 1 } } }
]
```

Notes - The actual compiler may choose either strategy based on simplicity; behavior is equivalent for most cases. - Null or missing productId yields product: null due to preserveNullAndEmptyArrays.

1.7.3. Nested arrays: expand(patient.visits[].diagnoses[])

Query

```
q = "expand(patient.visits[*].diagnoses[*]) &&
fields:[+_id,+patient.visits.diagnoses]"
```

Typical staged pipeline (high level)

```
1. $lookup patient -> __exp_patient; $set patient
2. $unwind visits (preserveNullAndEmptyArrays)
3. $unwind visits.diagnoses
4. $lookup diagnosis -> set visits.diagnoses[*]
5. $group back by root _id while reconstructing visits and diagnoses arrays in
original order
6. optional $project to constrain fields
```

Caveats - Depth and mixed object/array hops are bounded; traversal stops at a non-reference boundary. - For very large arrays, prefer targeted filtering before expansion to reduce fan-out.

1.7.4. When is AGGREGATION used vs FILTER?

- FILTER: no expand(...). Planner returns a Morphia Filter and the gateway executes find with dev.morphia.query.Query.
- AGGREGATION: one or more expand(...). Planner returns a pipeline with $lookup/$unwind and optional $project stages. Execution requires feature.queryGateway.execution.enabled=true; otherwise, POST /api/query/find returns 501. If any $lookup.from is "*unknown*", execution returns HTTP 422.