Filtering and Searching Specification

Filtering and searching operations are handled by the **Search Server**, a core component of the Parser ecosystem. The Search Server offers the following capabilities:

1. **Unified Query Interface**
   Enables querying across multiple underlying databases through a single, consistent API.
2. **Database-Agnostic Query Language**
   Supports a standard query format that abstracts away database-specific query syntax.
3. **Multi-Level Query Execution**
   Search operations can be performed at different levels:

   3.1 Filtering
   Executes a filter query across one or more databases. The system translates the unified filter into backend-specific queries and returns a set of matching results. This is useful for retrieving multiple records based on defined conditions.

   3.2 Filtering with Policy-Based Refinement
   Combines a filter query with a custom policy rule. Initially, the filter is applied across databases. The resulting dataset is then passed to a policy, which uses custom logic to refine the results further. The policy may return a subset of results or a single final match.

Filter and search specification:

Filter Spec Fields

Used to perform direct filtering queries across the system's databases.

Location: `body.values`

Field	Type	Required	Description
`matchType`	`string`	No	Defines the type of matching to be applied (e.g., exact, partial, semantic).
`filter`	`object`	Yes	Filtering logic to apply across one or more databases. *(Refer to next section for structure)*

Search Spec Fields

Used to apply filtering logic combined with a ranking/refinement policy.

Location: `body.values`

Field	Type	Required	Description
`matchType`	`string`	No	Defines the type of matching to be applied (e.g., exact, partial, semantic).
`rankingPolicyRule`	`object`	Yes (for policy-based search)	Configuration for applying a policy rule after initial filtering.
`rankingPolicyRule.policyRuleURI`	`string`	Yes	URI of the policy rule to apply for ranking or refining the filtered results.
`rankingPolicyRule.settings`	`object`	No	Policy-specific settings to influence its behavior.
`rankingPolicyRule.parameters.filterRule`	`object`	Yes	Filtering logic to apply before policy evaluation. *(Refer to next section)*
`rankingPolicyRule.parameters.return`	`number`	No	Number of top results to return after policy evaluation (e.g., top-K).

Filter query language:

The filter query is a declarative JSON structure used to define selection criteria across clusters, nodes, components, and other system resources. This structure is parsed and translated into backend-specific queries (e.g., MongoDB).

1. Condition Types

A filter query consists of one or more **conditions**. Each condition can be one of the following:

- **Simple Condition**: Defines a direct comparison on a field.
- **Logical Condition**: Combines multiple conditions using logical operators like `AND` and `OR`.

---

2. Simple Condition

A simple condition targets a specific field and compares it against a given value using one of the supported operators.

Structure:

```
{
  "variable": "<field_name>",
  "operator": "<comparison_operator>",
```

```
  "value": <comparison_value>
}
```

Supported Operators:

Operator	Description
`==`	Equal to
`IN`	Field value is in the provided list
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`LIKE`	Pattern match using wildcards (`*`)

---

3. Logical Condition

Logical conditions allow combining multiple simple or nested conditions using logical operators.

Structure:

```
{
  "logicalOperator": "AND" | "OR",
  "conditions": [
    { <simple_condition or logical_condition> },
    ...
  ]
}
```

Supported Logical Operators:

Operator	Description
`AND`	All sub-conditions must be true
`OR`	At least one sub-condition must be true

Filter Query Examples

All examples below refer to the following data structure (cluster document):

- Top-level fields like `id`, `regionId`, `status`, `memory`
- Nested structures like `gpus.memory`, `nodes.nodeData`, `config.policyExecutorId`

- Tags and lists like `tags`, `clusterMetadata.countries`

Example 1: Match by region

```
{
  "variable": "regionId",
  "operator": "==",
  "value": "us-west-2"
}
```

Example 2: Filter by reputation score

```
{
  "variable": "reputation",
  "operator": ">",
  "value": 90
}
```

Example 3: Tags include “ml” or “vision”

```
{
  "variable": "tags",
  "operator": "IN",
  "value": ["ml", "vision"]
}
```

Example 4: Memory greater than or equal to 128GB

```
{
  "variable": "memory",
  "operator": ">=",
  "value": 131072
}
```

Example 5: ID pattern match using LIKE

```
{
  "variable": "id",
  "operator": "LIKE",
  "value": "*vision*"
}
```

Example 6: Combine region and reputation using AND

```
{
  "logicalOperator": "AND",
```

```
  "conditions": [
    {
      "variable": "regionId",
      "operator": "==",
      "value": "us-west-2"
    },
    {
      "variable": "reputation",
      "operator": ">",
      "value": 90
    }
  ]
}
```

Example 7: Nested AND and OR conditions

```
{
  "logicalOperator": "AND",
  "conditions": [
    {
      "variable": "regionId",
      "operator": "==",
      "value": "us-west-2"
    },
    {
      "logicalOperator": "OR",
      "conditions": [
        {
          "variable": "tags",
          "operator": "IN",
          "value": ["ml"]
        },
        {
          "variable": "reputation",
          "operator": ">=",
          "value": 95
        }
      ]
    }
  ]
}
```

Example 8: Filter on nested field

```
{
  "variable": "gpus.memory",
```

```
  "operator": ">",
  "value": 49152
}
```

Use dot notation (`a.b.c`) to target nested fields in the document.

This filter query system enables flexible, expressive, and backend-agnostic filtering over rich, nested resource definitions. These queries can be combined with policy-based search flows or used independently for querying system metadata.

Using Filter Query Across Databases for Filtering

The Parser's filter engine supports flexible, declarative filtering across multiple data sources (e.g., cluster, block, vDAG, policy registries). It provides a unified query structure and abstraction over the underlying databases to retrieve, combine, and refine data based on user-defined rules.

---

1. Supported Entity Types

The following entity types are supported for filtering:

Entity Type	Description
`component`	Queries the component registry for registered system components.
`cluster`	Queries the cluster metadata database for cluster configurations.
`block`	Queries the block registry for deployed compute blocks.
`vdag`	Queries the virtual DAG (vDAG) registry.
`policy`	Queries registered policy definitions.
`policyFunction`	Queries registered policy functions.
`policyGraph`	Queries registered policy graphs.
`clusterMetrics`	Queries system-wide cluster-level metrics.
`blockMetrics`	Queries real-time block-level metrics.
`vdagMetrics`	Queries real-time vDAG-level metrics.

---

2. Query Paths and Composition

Filter queries can be composed across multiple related entities. In certain cases, filters must be evaluated in a specific order (e.g., filtering clusters before blocks). The following visual structure shows the **hierarchy and dependency flow** between queries:

Block Filtering Path

```
clusterQuery
    ↓
clusterQuery result → filter cluster.id
                            ↓
                    blockQuery
                            ↓
                    blockMetricsQuery (optional)
                            ↓
                       Final block list
```

Cluster Filtering Path

```
clusterMetricsQuery (optional)
         ↓
     clusterQuery
         ↓
   Final cluster list
```

vDAG Filtering Path

```
vdagMetricsQuery (optional)
         ↓
      vdagQuery
         ↓
   Final vDAG list
```

Component Filtering Path

```
componentQuery
     ↓
Final component list
```

Policy-Related Filtering Paths

```
policyQuery           → Final policy list
policyFunctionQuery   → Final policy function list
policyGraphQuery      → Final policy graph list
```

Each arrow (↓) indicates the flow of data filtering or dependency—i.e., the result of the upper query is used to refine the next layer of filtering.

- If a `clusterQuery` is provided before a `blockQuery`, the system will fetch matching clusters, extract their `id`s, and automatically add a condition to the `blockQuery` such as:

```
{
  "variable": "cluster.id",
  "operator": "IN",
  "value": [<cluster_ids>]
}
```

- Similarly, metrics queries like `blockMetricsQuery` or `clusterMetricsQuery` act as **pre-filters** for their corresponding entities.

---

vDAG Filtering

```
{
  "vdagQuery": { <filter_rule> },
  "vdagMetricsQuery": { <optional filter_rule on vDAG metrics> }
}
```

Policy Filtering

```
{
  "policyQuery": { <filter_rule> }
}
```

Policy Function Filtering

```
{
  "policyFunctionQuery": { <filter_rule> }
}
```

Policy Graph Filtering

```
{
  "policyGraphQuery": { <filter_rule> }
}
```

---

3. Execution Methods

Filter-Based Query Execution

Filter queries are executed using the `FilterSearch` class, which expects a parsed IR with the following structure:

```
{
  "matchType": "<entity_type>",
  "filter": {
    "<entity_type>Query": { <filter_rule> },
    ...
```

```
  }
}
```

Example:

```
{
  "matchType": "block",
  "filter": {
    "clusterQuery": {
      "variable": "regionId",
      "operator": "==",
      "value": "us-west-2"
    },
    "blockQuery": {
      "variable": "reputation",
      "operator": ">",
      "value": 80
    }
  }
}
```

Internally, the system evaluates dependent queries (e.g., filtering clusters before blocks) and builds a compound query using logical conjunctions (`AND`).

Policy-Based Search Execution

The `SimilaritySearch` class supports advanced filtering combined with post-filtering policy logic. It uses the following structure:

```
{
  "matchType": "<entity_type>",
  "rankingPolicyRule": {
    "policyRuleURI": "<policy_rule_uri>",
    "settings": { ... },
    "parameters": {
      "filterRule": { ... },   // Same structure as filter queries
      "return": <count>
    }
  }
}
```

- First, the `filterRule` is executed to narrow down the candidates.
- The results are then passed to the ranking policy for evaluation and further refinement.
- The policy can return a single best match or a ranked list of results.

Writing search policy:

Search policy can be combined with filtering to narrow down the search by using python program based filtering of results - which provides room for more advanced searching.

The policy that performs the advanced filtering must accept the filter results as input and return the new filter results.

Here is the sample policy structure that can be used for advanced filtering:

```
class AIOSv1PolicyRule:
    def __init__(self, rule_id, settings, parameters):
        self.rule_id = rule_id
        self.settings = settings
        self.parameters = parameters

    def eval(self, parameters, input_data, context):
       """
         inputs will be list of objects returned by the pre-filter query execution.
       """

       # Do the search here

       # return the new results
       return []
```

Example:

```
class AIOSv1PolicyRule:
    def __init__(self, rule_id, settings, parameters):
        self.rule_id = rule_id
        self.settings = settings
        self.parameters = parameters

    def eval(self, parameters, input_data, context):
        """
        Args:
            parameters (dict): Should include:
                - "minReputation" (int): minimum reputation required.
                - "region" (optional str): regionId to restrict the search to.
            input_data (list): List of cluster documents returned from pre-filter step.
            context (dict): Context for caching/sharing state across policies (not used here

        Returns:
            list: Filtered list of cluster objects.
        """
```

```
        min_reputation = parameters.get("minReputation", 90)
        target_region = parameters.get("region", None)

        result = []
        for cluster in input_data:
            if cluster.get("reputation", 0) >= min_reputation:
                if target_region:
                    if cluster.get("regionId") == target_region:
                        result.append(cluster)
                else:
                    result.append(cluster)

        return result
```

Here is the search payload:

```
{
  "matchType": "cluster",
  "rankingPolicyRule": {
   // policy is specified here
    "policyRuleURI": "policies.search.cluster-reputation-filter:v0.0.01-beta",
    "settings": {},
    "parameters": {
      "filterRule": {
        "matchType": "cluster",
        "filter": {
          "clusterQuery": {
            "logicalOperator": "AND",
            "conditions": [
              {
                "variable": "regionId",
                "operator": "==",
                "value": "us-west-2"
              },
              {
                "variable": "status",
                "operator": "==",
                "value": "live"
              }
            ]
          }
        }
      },
      "minReputation": 90
    }
  }
}
```

Executing the search using parser:

The search and filter can be executed using the parser's `executeAction` API and using either `filter` or `search` as the action.

Here is the `curl` example:

1. Search Request

```
curl -X POST http://<parser-host>:<port>/api/search \
  -H "Content-Type: application/json" \
  -d '{
    "header": {
      "templateUri": "Parser/V1",
      "parameters": {}
    },
    "body": {
      "values": {
        "matchType": "cluster",
        "rankingPolicyRule": {
          "policyRuleURI": "policies.search.cluster-reputation-filter:v0.0.01-beta",
          "settings": {},
          "parameters": {
            "filterRule": {
              "matchType": "cluster",
              "filter": {
                "clusterQuery": {
                  "logicalOperator": "AND",
                  "conditions": [
                    {
                      "variable": "regionId",
                      "operator": "==",
                      "value": "us-west-2"
                    },
                    {
                      "variable": "status",
                      "operator": "==",
                      "value": "live"
                    }
                  ]
                }
              }
            },
            "minReputation": 90
          }
```

```
      }
    }
  }
}'
```

---

2. Filter Request

```
curl -X POST http://<parser-host>:<port>/api/filter \
  -H "Content-Type: application/json" \
  -d '{
    "header": {
      "templateUri": "Parser/V1",
      "parameters": {}
    },
    "body": {
      "values": {
        "matchType": "cluster",
        "filter": {
          "clusterQuery": {
            "logicalOperator": "AND",
            "conditions": [
              {
                "variable": "regionId",
                "operator": "==",
                "value": "us-west-2"
              },
              {
                "variable": "status",
                "operator": "==",
                "value": "live"
              }
            ]
          }
        }
      }
    }
  }'
```
