Alma Mater Studiorum · Università di Bologna

**Dipartimento di Informatica — Scienza e Ingegneria**
**Corso di Laurea Triennale in Informatica**

# Native Operators for the Efficient Query of Jolie Tree-like Values

**Relatore:**
**Dott. Saverio Giallorenzo**

**Presentata da:**
**Matteo Manuelli**

**Correlatore:**
**Dott. Claudio Guidi**

**Anno Accademico 2024/2025**
**Sessione di Marzo**

*Ad Annagiuli**etta**,*
*anche se fa sempre molto arrabbiare tutti!*

# Table of Contents

# Chapter 1

## Introduction

Distributed systems are ubiquitous in modern computing, from household devices to global cloud platforms. Programming these systems introduces challenges beyond local programming: services must communicate across network boundaries, handle heterogeneous protocols, and process structured data exchanged in formats like JSON and XML.

Jolie is a programming language designed specifically for service-oriented distributed systems. Its distinctive feature is *protocol independence*: business logic remains unchanged whether services communicate via HTTP, SOAP, or binary protocols. This separation between behavior and deployment simplifies integration across heterogeneous systems.

Jolie realizes protocol independence through *tree-structured variables* whose structure is compatible with the data exchanged between services. Every variable in Jolie is a tree that can hold a root value and arbitrarily nested children. This design eliminates the impedance mismatch between the hierarchical data formats services exchange (JSON, XML) and the language's native data model—no serialization libraries or schema bindings required.

However, while Jolie's tree variables elegantly represent structured data, the language provides no native primitives for *querying* these structures. Since the language doesn't give native support to structuring queries on data, developers must write nested loops to traverse and filter tree data. TQuery, an external library, addressed this gap by introducing MongoDB-style operators, but moving the data between the main program and the library and the deep cloning semantics of some of its operators result in significant memory overhead—benchmarks show 10× memory consumption compared to imperative loops.

This thesis presents **PATHS** and **VALUES**, native Jolie language primitives for declarative tree querying. By directly exploiting Jolie's internal tree representation, these primitives eliminate both the verbosity of imperative traversal and the performance penalties of external libraries. A query that previously required nested loops or incurred substantial memory duplication becomes a single lightweight declarative expression.

The remainder of this thesis is organized as follows. Section 2 provides essential background on Jolie's architecture, tree variable model, and the TQuery library's limitations. Section 3 presents the PATHS and VALUES primitives, their syntax and semantics. Section 4 introduces the native `path` type and the `pval()` function operator for dereferencing path values. Section 5 summarizes contributions and discusses future work.

# Chapter 2

# Background: The Jolie Programming Language

This chapter provides the essential background for understanding the contributions of this thesis. We begin by examining the fundamental challenges of distributed systems programming—heterogeneity, fault handling, and evolution—and the verbosity they introduce in traditional languages. We then introduce Jolie, a service-oriented programming language designed to address these challenges through protocol independence and tree-structured variables. The chapter explores how Jolie's architecture separates communication protocols from business logic, and how its tree variable model eliminates impedance mismatch with hierarchical data formats. Finally, we examine existing approaches to querying tree-structured data, focusing on the TQuery library and the performance limitations that arise from moving data between a program and the library and its deep cloning semantics, motivating the need for native language primitives.

## 2.1 Challenges in Distributed Systems Programming

### 2.1.1 What & Why Distributed Systems?

A distributed system is a network of software components that communicate by exchanging messages. Service-Oriented Computing (SOC) structures these distributed systems around *services*—independent applications that offer operations and communicate through message passing. This paradigm draws a natural parallel with Object-Oriented Programming:

| Service-Oriented | Object-Oriented |
|---|---|
| Services | Objects |
| Operations | Methods |

Just as objects encapsulate state and expose methods, services encapsulate functionality and expose operations. The key difference: services communicate across process and network boundaries via message passing, not method calls.

These systems are ubiquitous at every scale: from microservices architectures to cloud platforms. However, programming these systems introduces challenges that compound with those of local programming. Despite their complexity, distributed systems offer compelling advantages:

- **Scalability**: Workloads distribute across multiple machines, scaling horizontally as demand grows.
- **Fault tolerance**: Redundant services continue operating when individual components fail.
- **Geographic distribution**: Services deployed close to users reduce latency.
- **Organizational alignment**: Different teams independently develop and deploy services that integrate through well-defined interfaces.
- **Technology heterogeneity**: Each service uses the most appropriate technology for its task—Python for machine learning, Rust for performance-critical components, Java for business logic.

These benefits explain why modern architectures—from microservices to cloud platforms—embrace distribution despite its inherent complexity.

### 2.1.2 The Verbosity of Low-Level Communications

This section demonstrates how even basic communication operations require extensive boilerplate code for proper error handling and resource management, both on the client and server side.

Consider one of the most basic distributed operation: sending data over a TCP socket. A naive Java implementation:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://someurl.com", 80));
Buffer buffer = . . .; // byte buffer
while (buffer.hasRemaining()) {
    channel.write(buffer);
}
```

This code is incomplete—it handles neither exceptions nor resource cleanup. A correct implementation requires extensive boilerplate:

```
SocketChannel socketChannel = SocketChannel.open();
try {
    socketChannel.connect(new InetSocketAddress("http://someurl.com",
80));
    Buffer buffer = . . .;
    while (buffer.hasRemaining()) {
        channel.write(buffer);
    }
}
catch (UnresolvedAddressException e) { . . . }
catch (SecurityException e) { . . . }
/* . . . many catches later . . . */
catch (IOException e) { . . . }
finally { channel.close(); }
```

Server-side code is even more complex. Handling asynchronous events on a channel requires selectors, key registration, and manual event dispatch:

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
    int readyChannels = selector.select();
    if(readyChannels == 0) continue;
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        if(key.isAcceptable()) { /* connection accepted */ }
        else if (key.isConnectable()) { /* connection established */ }
        else if (key.isReadable()) { /* ready for reading */ }
        else if (key.isWritable()) { /* ready for writing */ }
        keyIterator.remove();
    }
}
```

And this still omits exception handling. Further questions arise: What if different operations share the same channel? What data format for transmission? How to validate message types? How to change protocols after deployment?

### 2.1.3 Core Challenges

In addition to verbosity, distributed systems face fundamental challenges that arise from coordinating multiple independent services:

**Heterogeneity.** Different applications within the same system may use different communication mediums (Bluetooth, TCP/IP), different data protocols (HTTP, SOAP, XML-RPC), or different versions of the same protocol (SOAP 1.1 vs 1.2). For example, an internal high-performance service might use raw TCP with a binary protocol for speed, while a public-facing API uses HTTP with JSON for browser compatibility. Integrating these systems requires translation layers that understand both protocols—code that is tedious to write and maintain, and error-prone to maintain.

**Faults.** Distributed transactions span multiple services that can fail independently. Consider a purchase: a client contacts a store, the store requests payment from a bank, the client authenticates with the bank, and finally the store delivers goods. At any step, a service may be offline, a payment may be rejected, or delivery may fail—each requiring coordinated recovery.

**Evolution.** Distributed systems evolve independently. Each service may be maintained by different teams or organizations. Updates may introduce incompatible protocols, changed interfaces, or new behavioral requirements (e.g., adding authentication). Without careful design, evolution breaks existing integrations.

## 2.2 Jolie: A Service-Oriented Approach

Jolie is a service-oriented programming language designed to address the challenges of distributed systems programming through native support for Service-Oriented Computing. Jolie embodies SOC through three fundamental principles:

1. **Everything is a service**: The basic computational unit is a service, not a class or function.
2. **Services offer operations**: Each service exposes operations that define its interface.
3. **Services invoke operations**: Services interact by sending messages to operations offered by other services.

### 2.2.1 Hello World in Jolie

Even a minimal Jolie program demonstrates these principles:

```
from console import Console

service Main {
    embed Console as Console

    main {
        println@Console("Hello, world!")()
    }
}
```

This program embeds the `Console` service and invokes its `println` operation. The syntax `operation@Service(request)(response)` makes service invocation explicit—every interaction is a message exchange between services.

### 2.2.2 Protocol Independence: A First Glimpse

Jolie's distinctive feature is *protocol independence*. Consider a calculator service exposed over HTTP with JSON:

```
service Calculator {
    inputPort CalculatorPort {
        Location: "socket://localhost:8000"
        Protocol: http { .format = "json" }
        Interfaces: CalculatorInterface
    }
    main {
        [add(request)(response) {
            response = request.x + request.y
        }]
    }
}
```

To switch to raw TCP with Jolie's native binary protocol (SODEP), only the port declaration changes:

```
inputPort CalculatorPort {
    Location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: CalculatorInterface
}
```

The business logic remains untouched. No code changes, no recompilation—just a declaration change. This separation between *what* a service does and *how* it communicates is fundamental to Jolie's design.

## 2.3 Solving Protocol Heterogeneity

Distributed systems commonly integrate services that communicate via heterogeneous protocols—not every service uses a unique protocol, but the system as a whole must handle multiple incompatible communication standards. This heterogeneity arises from various factors: legacy systems, different organizational standards, or independent development by separate teams. Different applications within the same system may use different communication mediums (Bluetooth, TCP/IP), different data protocols (HTTP, SOAP, XML-RPC), or even different versions of the same protocol (SOAP 1.1 vs 1.2).

A common scenario involves integrating services where one service exposes its API using HTTP with JSON serialization, while another uses HTTP with XML serialization. Critically, these services cannot be easily modified—they may be legacy systems, third-party services, or systems with existing clients that depend on their current protocol.

Consider an organization requiring a unified interface to integrate:
- An **Employee Service** (HTTP + JSON, port 8001)
- A **Payroll Service** (HTTP + XML, port 8002)

The adapter must expose both functionalities through a single, consistent API while handling protocol translation transparently.

### 2.3.1 Traditional Approach: Protocol Adapters

In conventional programming languages, protocol translation necessitates the implementation of wrapper services. A Python-based adapter exemplifies this pattern:

```
@app.route('/payroll')
def get_payroll():
    emp_id = request.args.get('id', '1')

    # Call XML service
    response = requests.get(f"http://localhost:8002/payroll?
id={emp_id}")

    # Parse XML
    root = ET.fromstring(response.text)

    # Extract and convert to JSON
    return jsonify({"salary": int(root.find('salary').text)})
```

This code implements a **protocol adapter**—a software component that encapsulates protocol-specific operations (XML parsing, data extraction, JSON serialization) to bridge the impedance mismatch between heterogeneous systems. While functionally correct, this approach conflates business logic with deployment concerns, resulting in code that is tightly coupled to specific protocol implementations and requires modification whenever protocol details change.

### 2.3.2 Jolie's Approach: Declarative Protocol Separation

Jolie addresses this coupling through a fundamental language design principle: **separation of behavior from deployment** [1, p. 81]. As stated in the foundational work on Jolie, "the behaviour and deployment of a Jolie program are orthogonal: they can be independently defined and recombined as long as they have compatible typing" [1, p. 81]. In Jolie, business logic and communication protocols are specified in distinct, orthogonal language constructs.

The equivalent adapter in Jolie demonstrates this separation:

```
service AdapterService {
    inputPort AdapterAPI {  // HOW we transmit
        Location: "socket://localhost:8000"
        Protocol: http { .format = "json" }
        Interfaces: EmployeeInterface, PayrollInterface
    }
    outputPort EmployeeService {
        Location: "socket://localhost:8001"
        Protocol: http {
            .method = "get"; .format = "json";
            .osc.getEmployee.alias = "/employee?id=%{id}"
        }
        Interfaces: EmployeeInterface
    }
    outputPort PayrollService {
        Location: "socket://localhost:8002"
        Protocol: http {
            .method = "get"; .format = "xml";
            .osc.getPayroll.alias = "/payroll?id=%{id}"
        }
        Interfaces: PayrollInterface
    }
    main {  // WHAT we transmit
        [getEmployee(request)(response) {
            getEmployee@EmployeeService(request)(empData);
            response << empData
        }]
        [getPayroll(request)(response) {
            getPayroll@PayrollService(request)(payrollData);
            response << payrollData
        }]
    }
}
```

### 2.3.3 Key Observations

**Separation of Transmission from Logic**: Protocol specifications (HOW we transmit) are isolated in port declarations, while business logic (WHAT we transmit) resides in `main`. Modifying protocols requires no changes to business logic.

**Declarative Protocol Configuration**: The `Protocol` blocks are pure declarations—data formats, communication mediums, endpoint locations. No imperative parsing, serialization, or conversion code.

**Automatic Protocol Translation**: The Jolie runtime handles all protocol operations transparently: serialization to target formats (JSON/XML), deserialization from responses, and data structure mapping according to interface types.

**Type Safety and Error Handling**: The runtime automatically validates both incoming and outgoing data against interface types, detecting type mismatches and protocol violations in both directions without explicit validation code in the business logic.

**Location and Protocol Transparency**: Changing from `localhost:8002` to a remote server, or from HTTP to SOAP, requires only updating port declarations. Business logic remains completely unchanged.

## 2.4 Solving Data Heterogeneity: Tree-Structured Variables

Beyond protocol heterogeneity, distributed systems face data heterogeneity—the variety of hierarchical formats (JSON, XML, YAML) used to structure messages. Jolie's tree-structured variable model addresses this challenge through a domain-specific design.

> "Last year I went to a conference hosted by Microsoft called Lang.NEXT, and I saw a number of talks, many of which were the leaders of a particular language talking about a new version that was coming out.
>
> I really was struck by one thing about these talks and these languages, which is most of the talks consisted of features being added by taking something from another language and adding it to this one. And I realized that what's happening is all of these languages are turning into the same language. [...] You want to have different languages for different problems. You want to have different domains be solved by different ways of thinking and different notations. In other words, you kind of want a tool that's optimized for each particular way you're working."
>
> — Rob Pike, dotGo 2015 [2]

### 2.4.1 Three Principles of Jolie's Variable Model

Jolie's variable model follows three principles:

1. **Scalar-Array Unification** (`a = a[0]`): No distinction between single values and arrays —accessing `a` equals accessing `a[0]`.
2. **Vivification**: Assigning to `x.y.z` automatically creates intermediate nodes `x` and `x.y` if they don't exist.

### 2.4.2 Safe Navigation Example

The following Jolie program demonstrates vivification—accessing non-existent fields returns empty values without exceptions:

```
from console import Console

service Main {
    embed Console as Console

    main {
        a = 42;

        // Access non-existent field in expression
        if (a.nonexistent == "") {
            println@Console("a.nonexistent equals empty string")();
            println@Console("No exception thrown - safe navigation!")()
        }
    }
}
```

Output:

```
a.nonexistent equals empty string
No exception thrown - safe navigation!
```

3. **Everything is a Tree**: Every variable, including array elements, is a tree with a root value and child nodes.

This design choice is motivated by the domain Jolie targets: service-oriented systems. Services communicate by exchanging structured data—JSON arrays, XML hierarchies, nested objects. These formats are inherently tree-structured with repeating elements. Traditional programming languages separate primitives, objects, and arrays into distinct type categories, creating an **impedance mismatch**: the data structures services exchange do not match the native data model of the language, requiring explicit serialization, deserialization, and schema binding. Moreover, traversing nested structures typically necessitates either custom traversal code or external query libraries (JSONPath, XPath, jq), introducing additional dependencies and data conversion overhead.

Jolie eliminates this mismatch by adopting trees as the universal data representation. A JSON object can be mapped into a Jolie tree. A JSON array maps to multiple tree instances under the same child name. There is no conversion layer, no object-relational mapping—the data format used for communication is the same data structure used in the program.

This uniformity yields several advantages for service integration:

**No Impedance Mismatch**: Data received from external services requires no transformation—it is already in the native format the language operates on.

**Protocol-Agnostic Logic**: Business logic manipulates trees. Whether those trees arrived as JSON, XML, or binary format is irrelevant—the code remains identical.

## 2.5 Querying Hierarchical Data: A Cross-Language Comparison

To illustrate the challenges of querying hierarchical data in traditional programming languages, consider a family tree dataset with the following structure:

```
type Person struct {
    Name     string   `json:"name"`
    Sex      string   `json:"sex"`
    Children []Person `json:"children"`
}
```

This represents a nested tree where each `Person` has a name, sex, and zero or more children, who themselves may have children (grandchildren from the original person's perspective). Example data:

```
[
  {
    "name": "John Smith",
    "sex": "Male",
    "children": [
      {
        "name": "Alice Smith",
        "sex": "Female",
        "children": [
          { "name": "John Smith", "sex": "Male", "children": [] }
        ]
      }
    ]
  }
]
```

**Query Goal:** Find all grandfathers (male persons) whose name matches at least one of their grandchildren. This requires:

1. Filtering to male persons only
2. Traversing two levels of nesting (person → children → grandchildren)
3. Checking if any grandchild's name matches the grandfather's name
4. Collecting matching results

Each language implements this query differently:

**Go** (19 lines): Implements the query imperatively with explicit nested loops and manual control flow. Requires manual iteration state management and explicit result collection using `append()`.

```go
func findMatchingGrandfathers(persons []Person) []Person {
    var result []Person

    for _, person := range persons {
        if person.Sex != Male {
            continue
        }

        for _, child := range person.Children {
            for _, grandchild := range child.Children {
                if grandchild.Name == person.Name {
                    result = append(result, person)
                }
            }
        }
    }

    return result
}
```

**Rust** (12 lines): Uses functional iterator combinators (`filter`, `flat_map`, `any`) to express the query declaratively. The iterator chain avoids explicit loops, but still requires manual flattening of nested structures and complexity in expressing the "grandchild matches name" condition across closure boundaries.

```rust
fn find_matching_grandfathers(persons: &[Person]) -> Vec<&Person> {
    persons
        .iter()
        .filter(|person| person.sex == Sex::Male)
        .filter(|person| {
            person.children
                .iter()
                .flat_map(|child| child.children.iter())
                .any(|grandchild| grandchild.name == person.name)
        })
        .collect()
}
```

**JavaScript** (9 lines): Leverages array methods (`filter`, `flatMap`, `some`) for a functional approach similar to Rust. The `flatMap` operation flattens the two-level nesting (children → grandchildren) in a single step, and `some` checks for name matches. However, the nested anonymous functions and method chaining can obscure the query logic.

```
function findMatchingGrandfathers(persons) {
    return persons
        .filter(person => person.sex === 'Male')
        .filter(person =>
            person.children
                .flatMap(child => child.children)
                .some(grandchild => grandchild.name === person.name)
        );
}
```

**Ballerina** (9 lines): Uses language-integrated query expressions (LINQ-style) with `from`/`where`/`select` clauses. The nested query expression checks for matching grandchildren, but requires computing the length of the result set to determine if any match exists, which is less direct than checking existence.

```
function findMatchingGrandfathers(Person[] persons) returns Person[] {
    return from var person in persons
        where person.sex == MALE
        where (from var child in person.children
               from var grandchild in child.children
               where grandchild.name == person.name
               select 1).length() > 0
        select person;
}
```

**Jolie (Pre-TQuery)**:

```
from console import Console
from file import File

service Main {
    embed Console as Console
    embed File as File

    main {
        readFile@File({
            filename = "test-data.json"
            format = "json"
        })(data);

        for (person in data._) {
            if (person.sex == "Male") {
                for (child in person.children) {
                    for (grandchild in child.children) {
                        if (grandchild.name == person.name) {
                            println@Console("Found: " + person.name)()
                        }
                    }
                }
            }
        }
    }
}
```

Output:

```
Found: John Smith
Found: George Brown
```

Despite Jolie's tree-structured variables eliminating the need for JSON parsing and object mapping—where JSON, XML, YAML, and other hierarchical formats map seamlessly to the same native data structure—querying still requires explicit nested loops similar to Go. Without declarative query capabilities, developers are forced to write the same three-level traversal pattern to traverse and filter tree data. TQuery addresses this gap by providing declarative operators that leverage Jolie's tree abstraction.

## 2.6 The Query Gap: From Imperative Loops to Declarative Queries

TQuery introduces declarative tree querying to Jolie by integrating MongoDB aggregation framework operators directly into the language [3]. This integration addresses a critical challenge in modern distributed systems: ephemeral data handling. In edge and fog computing scenarios—where data privacy regulations (GDPR, HIPAA) mandate minimal data retention, or where local processing reduces network overhead—using external databases like MongoDB introduces unnecessary dependencies, security risks, and performance penalties. TQuery eliminates these issues by enabling powerful query operations on Jolie's native tree structures without requiring external database systems.

By treating trees as first-class queryable structures, TQuery allows developers to express complex data traversal and filtering as declarative query pipelines rather than nested imperative loops. Operations such as `unwind` (flatten nested arrays), `match` (filter records), `project` (shape output), `group` (aggregate data), and `lookup` (join trees) transform what would be four-level nested loops into single-line declarative expressions, while maintaining Jolie's protocol-agnostic design and type safety.

#### 2.6.0.1 Example: Finding Namesake Grandfathers

Consider the task of finding all grandfathers whose name matches at least one of their grandchildren. The imperative approach requires three nested loops with conditional checks:

```
for (person in data._) {
    if (person.sex == "Male") {
        for (child in person.children) {
            for (grandchild in child.children) {
                if (grandchild.name == person.name) {
                    result[#result] << person
                }
            }
        }
    }
}
```

Listing 1: Imperative grandfather query with nested loops

With TQuery, this complexity collapses into a declarative pipeline with three stages:

```
// TQuery Pipeline: unwind → filter sex → match names
stages[0].unwindQuery = "_.children.children";
stages[1].matchQuery.equal << { path = "_.sex" data = "Male" };
stages[2].matchQuery.equal << { left = "_.name" right =
"_.children.children.name" };

pipeline@TQuery({ data << data pipeline << stages })(filtered);
```

Listing 2: TQuery grandfather query with declarative pipeline

The `unwind` operator flattens the nested `children.children` structure, eliminating two levels of iteration. The first `match` filters by sex, replacing the conditional check. The second `match` performs path-to-path comparison, testing equality between `_.name` (grandfather) and `_.children.children.name` (grandchild) within each unwound record. This reduces 18 lines of imperative code to 5 lines of declarative queries—a 72% reduction while improving readability and maintainability.

#### 2.6.0.2 Performance Considerations: The Cost of Context Preservation

TQuery's declarative elegance comes at a significant performance cost due to transferring data from a Jolie program to the library for querying and due to some of its operators' se-

mantics. For instance, the TQuery specification [3] explicitly defines the unwind operator's semantics:

> The unwind operator ω takes as inputs an array and a path p. The result of the application is a new array containing the "unfolding" of the input array under the path, i.e., where we take each element e from the input array, we find all values under p in e and, for each value, we include in the new array **a copy of e** except it holds only that single value under p.
>
> — TQuery specification [3], "An overview of the Tquery operators"

The critical passage is "**a copy of e**"—for each unwound value, TQuery creates a complete copy of the parent element, including all sibling fields. Consider a realistic e-commerce scenario:

```
order.items[0] = { productId: "SKU-001", productName: "Wireless Mouse",
quantity: 2 }
order.items[1] = { productId: "SKU-002", productName: "USB-C Cable",
quantity: 1 }
order.items[2] = { productId: "SKU-003", productName: "Screen
Protector", quantity: 5 }

order.auditLog[0..9] = [
  { event: "order_created", userId: "user123", productId: "SKU-001" },
  { event: "payment_processed", amount: 109.97, productId: "SKU-002" },
  { event: "inventory_reserved", warehouse: "WH-NA-01", productId:
"SKU-003" },
  // ... 7 more audit events
]

unwind@TQuery({ query = "order.items" })
```

When unwinding `items`, the specification requires creating "a copy of e" (the entire order) for each item. This means the 10-element `auditLog` array gets cloned three times—once for each item—resulting in substantial memory duplication.

This behavior is not an implementation artifact but a consequence of the formal semantics. The specification ensures that when unwinding a path, all sibling fields are preserved in each resulting record, requiring complete materialization of the "context" for every unwound element.

**Projection does not eliminate the multiplicative copying.** One might attempt to reduce memory usage by projecting away unnecessary fields before unwinding—retaining only `productId` and `event` from `auditLog` while discarding `productName`, `quantity`, and other attributes. However, this optimization cannot avoid the fundamental issue: unwinding `items` produces 3 output records (one per item), and each output record contains a complete copy of the entire `auditLog` array with its 10 entries. The result is $3 \times 10 = 30$

individual record copies. While projection reduces the size of each copy, the multiplicative duplication—inherent to the unwind semantics—remains unavoidable.

PATHS and VALUES eliminate this overhead by operating directly on Jolie's tree-structured values without intermediate transformations. The `values` expression returns matching elements with zero-copy evaluation, while `paths` preserves hierarchical context through path references (e.g., `data.companies[0].departments[1].projects[3]`)—enabling queries to identify not just matching leaf nodes but also their position within the parent hierarchy, without verbose joins or intermediate copies.

#### 2.6.0.3 Empirical Validation: Benchmark Results

To quantify the performance implications of the above overhead, we present benchmarks comparing two approaches for filtering deeply nested data: traditional imperative loops and TQuery pipelines.

**Dataset Structure:** The test data (`large_data.json`) contains 4,800 projects nested 4 levels deep following the hierarchy: companies → departments → teams → projects. The dataset is generated with 60 companies, each containing 5 departments, each with 4 teams, and each team managing 4 projects. This structure mirrors real-world organizational hierarchies commonly found in enterprise systems.

**Test Query:** Each benchmark filters projects matching two conditions across different nesting levels: `status == "in_progress"` AND `technology == "Python"`. This query pattern is representative of common analytical tasks where related fields at the same nesting level must be checked simultaneously.

**Implementations Compared:** Two approaches process the same query:

**Imperative**: Traditional nested for-loops traversing all four levels with conditional checks:

```
// WITHOUT TQUERY: Nested loops
resultCount = 0;
for (c = 0, c < #global.data.companies, c++) {
    company -> global.data.companies[c].company;
    for (d = 0, d < #company.departments, d++) {
        dept -> company.departments[d];
        for (t = 0, t < #dept.teams, t++) {
            team -> dept.teams[t];
            for (p = 0, p < #team.projects, p++) {
                project -> team.projects[p];
                if (project.status == request.status) {
                    hasTech = false;
                    for (tech = 0, tech < #project.technologies && !
hasTech, tech++) {
                        if (project.technologies[tech] ==
request.technology) {
                            hasTech = true
                        }
                    };
                    if (hasTech) {
                        response.results[resultCount] << project;
                        resultCount++
                    }
                }
            }
        }
    }
}
```

**TQuery**: Declarative pipeline using `unwind` on the path, followed by `match` with AND conditions, and `project` to shape output:

```
// WITH TQUERY: Pipeline with unwind + match + project
ps[0].unwindQuery =
"companies.company.departments.teams.projects.technologies";

ps[1] << {
    matchQuery.and << {
        left.equal << {
            path = "companies.company.departments.teams.projects.status"
            data = request.status
        }
        right.equal << {
            path =
"companies.company.departments.teams.projects.technologies"
            data = request.technology
        }
    }
};

ps[2] << {
    projectQuery[0] << {
        dstPath = "project_id"
        value.path =
"companies.company.departments.teams.projects.project_id"
    }
    projectQuery[1] << {
        dstPath = "name"
        value.path = "companies.company.departments.teams.projects.name"
    }
    projectQuery[2] << {
        dstPath = "status"
        value.path =
"companies.company.departments.teams.projects.status"
    }
    projectQuery[3] << {
        dstPath = "technologies"
        value.path =
"companies.company.departments.teams.projects.technologies"
    }
};

pipeline@TQuery({
    data << global.data
    pipeline << ps
})(filtered)
```

The `unwind` operator flattens the nested hierarchy, but creates a copy of the entire parent context for each unwound element, leading to the memory overhead discussed earlier.

**Benchmark Execution:** Tests measure concurrent request performance using a Python script that sends multiple parallel requests to separate Jolie services (one per implemen-

tation). Each service processes the same dataset and query. Tests run at three concurrency levels (5, 7, and 9 parallel requests) to observe performance under varying load. Metrics collected include P50/P95 latency percentiles, maximum heap memory usage, and garbage collection events extracted via `jstat` from the JVM runtime.

**Test Environment:** CPU: Intel(R) Core(TM) 7 150U, Memory: 15 GiB, OS: Debian GNU/Linux (kernel 6.1.0-40-amd64), Java: OpenJDK 64-Bit Server VM (build 21.0.8+9-Debian-1).

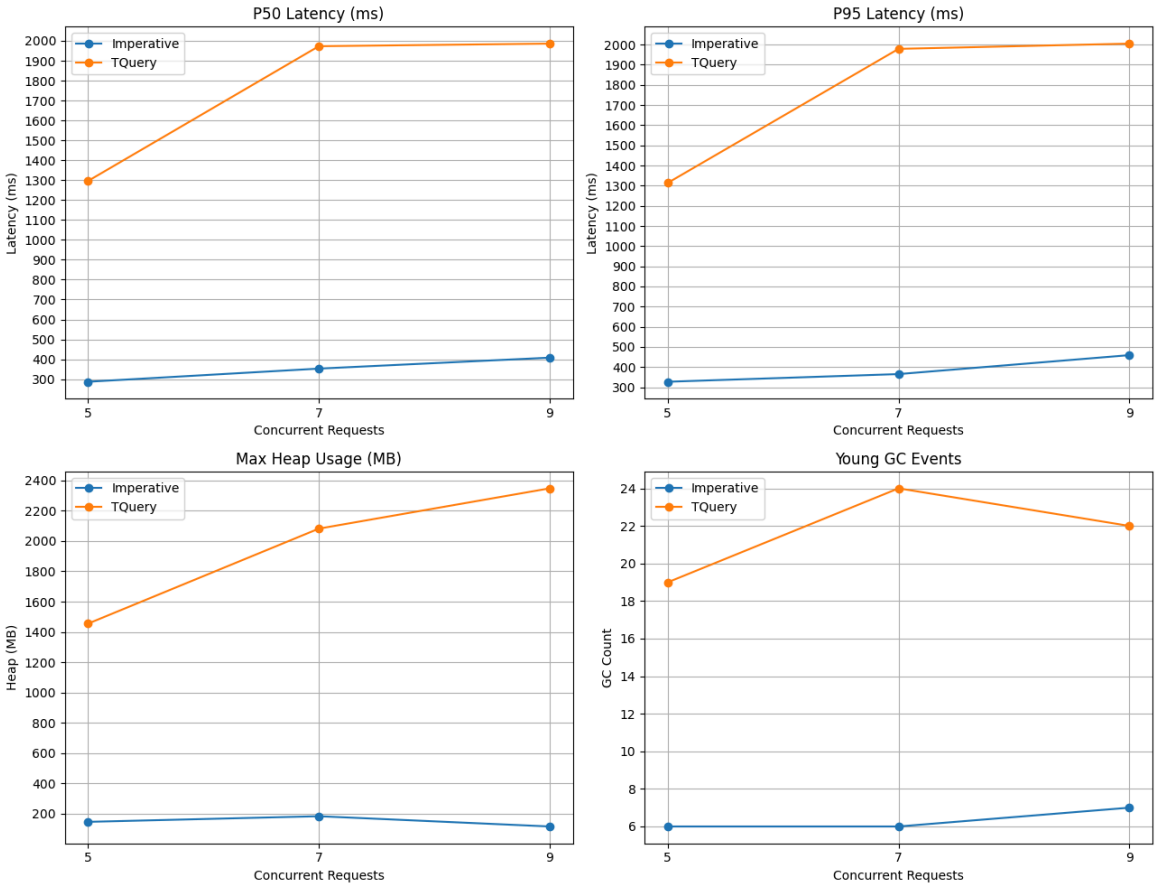


Figure 1: Performance comparison across different concurrency levels on 4,800 nested projects

Across all concurrency levels, TQuery consistently exhibits 4.5–5.6× higher latency and consumes approximately 10× more memory than the imperative approach. The elevated garbage collection activity confirms sustained memory pressure from materializing cloned contexts. These results validate the theoretical analysis: the specification's requirement for "a copy of e" translates directly into substantial runtime costs when processing hierarchical data with sibling arrays.

# Chapter 3

# Native Declarative Querying: PATHS and VALUES

This chapter presents the PATHS and VALUES primitives, native Jolie language constructs for declarative tree querying. We begin by introducing the syntax and semantics of these expressions, including the six path operations for navigating tree structures. We then examine the WHERE clause for filtering results and the $ operator for referring to the current candidate value. The HAS operator for structural filtering is presented, followed by a discussion of the language extensions required to support these primitives. Finally, we analyze the implementation's complexity, demonstrating linear time and space performance through zero-copy navigation of Jolie's internal tree representation.

## 3.1 Motivation

As established in the previous chapter, native Jolie provides no primitives to exploit its sophisticated tree variable representation for querying. While TQuery addressed this gap through an external library, it introduced significant performance overhead. The PATHS and VALUES expressions represent native Jolie language primitives that provide declarative tree querying by directly exploiting Jolie's internal Java tree variable representation (`Value` and `ValueVector` structures), eliminating both external dependencies and cloning overhead.

### 3.1.1 Motivating Example

By directly exploiting Jolie's internal Java tree variable representation, the grandfather problem—previously requiring nested loops or TQuery's pipeline with cloning overhead—simplifies dramatically. The same query that took 15 lines of imperative code or incurred substantial memory duplication with TQuery now becomes a single declarative expression:

```
values data._[*] where
    $.sex == "Male" &&
    $.name == $.children[*].children[*].name
```

Listing 3: PATHS/VALUES grandfather query with declarative filtering

This expression traverses the entire data tree, filters for male persons, and checks if any grandchild's name matches the grandfather's name—all in three lines without explicit loops or intermediate variables.

The query integrates naturally into a complete Jolie program:

```
from file import File
from console import Console

service Main {
    embed File as File
    embed Console as Console

    main {
        readFile@File({
            filename = "test-data.json"
            format = "json"
        })(data);

        result << values data._[*] where
            $.sex == "Male" &&
            $.name == $.children[*].children[*].name;

        println@Console("Found " + #result.results + " matching
grandfathers:")();
        i = 0;
        while (i < #result.results) {
            println@Console("- " +
                result.results[i].name + " " +
                result.results[i].surname
            )();
            i++
        }
    }
}
```

Running this program produces:

```
Found 2 matching grandfathers:
- John Smith
- George Brown
```

## 3.2 Syntax and Semantics

PATHS and VALUES expressions share a common syntax structure that enables flexible tree querying through composable operations.

### 3.2.1 Basic Syntax

```
paths <path-clause> where <where-clause>
values <path-clause> where <where-clause>
```

The **path-clause** defines which parts of the tree to traverse, while the **where-clause** filters results based on conditions. The key distinction: `paths` returns path-type values

representing locations in the tree (e.g., `data[0].field`), while `values` returns deep copies of the actual data at those locations. The native `path` type enables subsequent dereferencing via `pval()`, which provides reference access without copying.

### 3.2.2 Path Operations

Six composable operations enable flexible tree navigation:

| Operation | Syntax | Description |
|---|---|---|
| Field | `.field` | Access named property |
| Field Wildcard | `.*` | Match any field at current level |
| Array Index | `[n]` | Specific array element |
| Array Wildcard | `[*]` | All array elements |
| Recursive Field | `..field` | Find field at any depth |
| Recursive Wildcard | `..*` | All fields at any depth |

**Grammar:**

```
path-clause := identifier array-access? suffix*

suffix := dot-access array-access?

dot-access := '.' field
            | '.' '*'
            | '.' '.' field
            | '.' '.' '*'

array-access := '[' integer ']'
              | '[' '*' ']'
```

Where `identifier` is the root variable name, `field` is an identifier, and `integer` is a numeric literal. The grammar naturally prevents consecutive array accesses: each `array-access` is optional and can only appear after the identifier or after a `dot-access`, never after another `array-access`.

These operations compose naturally. For example:

- `data[0]` — first element of data array
- `data[*]` — all elements in data array
- `data[*].items[2]` — third item in each data record
- `data[*].items[*]` — all items in all data records
- `tree..status` — all status fields at any depth
- `data[*].*[*]` — all array elements in all fields of all data records

### 3.2.3 The WHERE Clause and $ Operator

The WHERE clause uses the special `$` operator to represent the current value being evaluated. When combined with path operations, `$` enables powerful filtering:

- `where $ > 5` — current value exceeds 5

- `where $.status == "active"` — current value has status field equal to "active"
- `where $.tags[*] == "urgent"` — any tag equals "urgent" (existential)
- `where $..technologies[*] == "Python"` — any technologies field at any depth contains "Python"
- `where $ has "field"` — current value contains field "field"

The `has` operator checks field existence, useful for structural filtering rather than value-based filtering.

**WHERE Clause Grammar:**

```
where-clause := 'where' boolean-expr

boolean-expr := or-expr

or-expr := and-expr
         | or-expr '||' and-expr

and-expr := not-expr
          | and-expr '&&' not-expr

not-expr := '!' not-expr
          | primary-expr

primary-expr := comparison-expr
              | '(' boolean-expr ')'

comparison-expr := operand compare-op operand
                 | operand 'has' operand

operand := current-value
         | literal
         | identifier

current-value := '$' suffix*

compare-op := '==' | '!=' | '<' | '>' | '<=' | '>='

literal := string-literal | integer | boolean
```

The grammar encodes operator precedence explicitly: `||` (lowest) binds less tightly than `&&`, which binds less tightly than `!` (highest), with parentheses overriding precedence. The `current-value` production allows `$` with all path operations—field access, wildcards, array indexing, and recursive descent (e.g., `$.field[*]`, `$..nested[*]`).

#### 3.2.3.1 Parse-Time Restriction of `$`

The `$` operator is restricted to WHERE clauses through parse-time validation. Attempting to use `$` outside a WHERE clause produces a compile-time error:

```
// INVALID: $ outside WHERE clause
x = $;  // Error: $ can only be used in WHERE clauses
```

The parser maintains an `inWhereClause` flag that is set when entering a WHERE clause and cleared upon exit. When the scanner encounters `$`, the parser checks this flag and rejects the program if `$` appears in an invalid context:

```
/home/matteo/examples/test.ol:15: error: $ can only be used in WHERE
clauses

15:         x = $;
               ^
```

This restriction ensures `$` has well-defined semantics—it always refers to the current candidate value during WHERE clause evaluation, never to an undefined or ambient context.

### 3.2.4 Existential Semantics

The WHERE clause uses **existential semantics**: a condition matches if **any** value in the evaluation satisfies it (∃), not all. This is critical for array comparisons.

When path operations with wildcards (`[*]`, `.*`) produce arrays, comparison operators perform element-wise existential matching: the condition succeeds if **any** element from the left operand matches **any** element from the right operand.

In the grandfather example:

```
$.name == $.children[*].children[*].name
```

The expression `$.children[*].children[*].name` expands to an array of all grandchildren names. The equality operator checks if **any** element from the left operand (the grandfather's name) equals **any** element from the right operand array (grandchildren names). This eliminates the need for explicit loops to check "does any element match?"

**Example:** If `$.children[*].children[*].name` yields `["John", "Emma"]`, the condition `$.name == $.children[*].children[*].name` with `$.name = "John"` succeeds because "John" appears in the array.

Similarly, in the companies filtering:

```
$.technologies[*] == "Python"
```

The left operand `$.technologies[*]` produces an array of all technologies for the current project. Internally, the comparison operator converts both operands to arrays: the left side yields multiple values through the wildcard, while the right side literal `"Python"` becomes a singleton array `["Python"]`. The operator then performs a nested iteration over both arrays, succeeding if **any** element from the left matches **any** element from the right. This uniform array-based evaluation applies to all comparison operators—whether

comparing array-to-literal (`$.technologies[*] == "Python"`), literal-to-literal (`$.status == "active"`), or array-to-array (`$.children[*].name == $.grandchildren[*].name`)—enabling consistent existential semantics across all operand combinations.

#### 3.2.4.1 Independent Condition Evaluation

Each comparison in a WHERE clause evaluates independently to a boolean, which are then combined by logical operators (&&, ||). Consider querying for employees named "Matteo Rossi":

```
values data where
    $.employees[*].name == "Matteo" &&
    $.employees[*].surname == "Rossi"
```

If the data contains:

- Employee 1: name="Matteo", surname="Maggio"
- Employee 2: name="Roberto", surname="Rossi"

The query **matches** even though no single employee satisfies both conditions:

1. `$.employees[*].name == "Matteo"` evaluates independently → TRUE (Matteo Maggio exists)
2. `$.employees[*].surname == "Rossi"` evaluates independently → TRUE (Roberto Rossi exists)
3. TRUE && TRUE → TRUE

Each comparison produces a boolean result based on its own existential evaluation, without correlation to which specific elements satisfied other comparisons. The logical operators combine these boolean results, not the underlying array elements.

To match a specific employee, reference the same structural level:

```
// Match specific employee
values data.employees[*] where $.name == "Matteo" && $.surname ==
"Rossi"
```

Here, `$` refers to each employee candidate individually, so both conditions evaluate against the same employee object.

### 3.2.5 The HAS Operator

Standard comparison operators (==, <, etc.) filter based on **values**—they compare the content stored at a path. However, real-world data often requires **structural** filtering: selecting nodes based on which fields they contain, regardless of the values in those fields.

Consider a common API design challenge: optional fields with semantic meaning. When designing a meeting request API, clients may send:

```
{ "title": "Sprint Planning", "participants": ["Alice", "Bob"] }
{ "title": "Solo Review" }
{ "title": "Team Sync", "participants": [] }
```

The second request has **no** `participants` field—intuitively meaning "participants not specified." The third has an **empty** `participants` array—meaning "explicitly no participants." These are semantically different: one is unspecified, the other is explicitly empty.

Value-based comparison cannot distinguish these cases. Checking `$.participants == ""` or array length fails when the field is absent entirely. The `has` operator addresses this:

```
// Find meetings where participants were explicitly specified
specified << values requests[*] where $ has "participants"
```

This returns only the first and third requests, filtering based on field **existence** rather than field **value**. Without `has`, defensive programming is required:

```
// Without has: manual existence check
if (is_defined(request.participants)) {
    // process participants
}
```

The `has` operator takes a field name to check:

```
// Filter requests that specify participants
values requests[*] where $ has "participants"

// Combine with value comparisons
values orders[*] where $ has "priority" && $.priority == "high"

// Check nested structure existence
values users[*] where $.profile has "preferences"
```

This structural filtering complements value-based filtering, enabling queries that distinguish "field absent" from "field present but empty"—a distinction that would otherwise require explicit null-checking or try-catch patterns in imperative code.

## 3.3 Language Extensions

Implementing PATHS and VALUES required extending Jolie's lexical and syntactic infrastructure.

### 3.3.1 New Keywords and Tokens

The scanner recognizes six new tokens:

| Token | Purpose |
|---|---|
| `paths` | Begins a PATHS expression |
| `values` | Begins a VALUES expression |
| `where` | Introduces the filter clause |
| `has` | Structural existence operator |
| `pval` | Path evaluation function |

| Token | Purpose |
|---|---|
| `$` | Current value reference in WHERE |

These tokens are registered as unreserved keywords, meaning they can still be used as identifiers in contexts where there is no ambiguity—preserving backward compatibility with existing Jolie programs that may use these names as variables.

## 3.4 Complexity Analysis

The implementation achieves linear time and space complexity through careful design choices.

### 3.4.1 Time Complexity

The `navigate()` function applies operations sequentially, transforming a set of candidate locations at each step:

```
List<Candidate> candidates = new ArrayList<>();
candidates.add(new Candidate(rootVec, from.path()
[0].key().evaluate().strValue()));

for (PathOperation op : ops) {
    List<Candidate> next = new ArrayList<>();
    candidates.forEach(c -> next.addAll(expand(c, op)));
    candidates = next;
}
```

Each `Candidate` is a lightweight record storing a reference (vector + index) and path string—no value copying occurs. The variable `ops` contains the list of path operations. For example, `data.items[0].name` produces `ops = [Field("items"), ArrayIndex(0), Field("name")]`.

The nested loop structure suggests O(|ops| × |candidates|) complexity. However, |candidates| is not the tree size N—it represents the current working set of locations being navigated. For non-recursive operations, |candidates| is bounded by tree width w (the maximum branching factor encountered), not total tree size. The complexity is thus O(|ops| × w), which is linear in the tree structure being traversed.

Consider how this processes the query `data[*].status` over an array with 5 elements:

```
Tree structure:
data[0].status = "ok"
data[1].status = "pending"
data[2].status = "ok"
data[3].status = "failed"
data[4].status = "pending"

ops = [ArrayWildcard(), Field("status")]

Initial:
  candidates = [(ref: data, path: "data")]

Step 1: Apply ArrayWildcard()
  Process 1 candidate (data) → produces 5 candidates
  candidates → [(ref: data[0], path: "data[0]"),
                (ref: data[1], path: "data[1]"),
                (ref: data[2], path: "data[2]"),
                (ref: data[3], path: "data[3]"),
                (ref: data[4], path: "data[4]")]
  Work: O(5)

Step 2: Apply Field("status")
  Process 5 candidates → produces 5 candidates
  (ref: data[0], "data[0]") → (ref: data[0].status, "data[0].status")
  (ref: data[1], "data[1]") → (ref: data[1].status, "data[1].status")
  (ref: data[2], "data[2]") → (ref: data[2].status, "data[2].status")
  (ref: data[3], "data[3]") → (ref: data[3].status, "data[3].status")
  (ref: data[4], "data[4]") → (ref: data[4].status, "data[4].status")
  Work: O(5)

Total work: O(1 + 5 + 5) = O(11)
Total candidates processed: 1 + 5 + 5 = 11
Complexity: O(|ops| × w) where w = 5 (array size)
```

Visualization of the transformation:

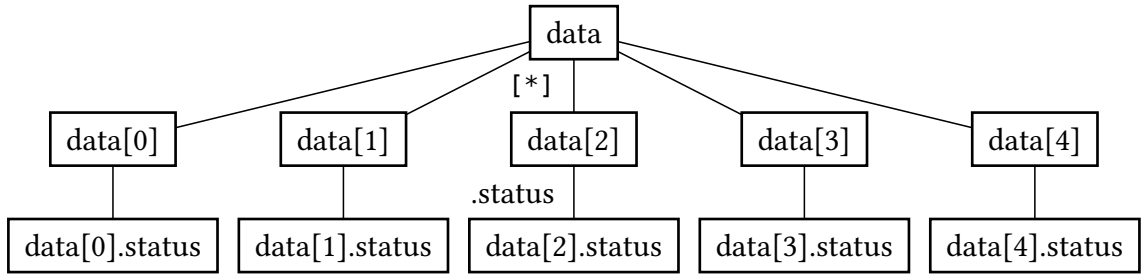


Figure 2: Linear traversal: each node visited once per operation, references only

Each operation transforms the current candidate set by creating new references, never copying tree data.

Work per operation type (per candidate):

- **Field access** (`.field`): O(1) via hash map lookup

- **Array index** (`[n]`): O(1) via direct access
- **Field wildcard** (`.*`): O(k) where k = number of fields in that node
- **Array wildcard** (`[*]`): O(m) where m = array size of that node
- **Recursive descent** (`..field`, `..*`): O(N) via breadth-first search over the entire subtree

For non-recursive operations, the total complexity is O(|ops| × w) where w is the maximum width (branching factor) encountered during traversal—bounded by tree structure, not tree size N.

For recursive operations (`..field`, `..*`), the single operation itself performs a breadth-first search over all N nodes, dominating the complexity: O(N) regardless of |ops|.

### 3.4.2 Space Complexity

Navigation creates `Candidate` wrapper objects without copying values:

```
// Candidate stores: (ValueVector reference, index, path string)
record Candidate(ValueVector vector, UnsignedInteger index, String path)
{
    Value value() { return vector.get(index.intValue()); }
}

// Field access: return reference wrapper, no cloning
ValueVector vec = c.value().children().get(name);
yield List.of(new Candidate(vec, c.path() + "." + name));
```

For N matching nodes, space complexity is O(N × d) where d is tree depth:
- N `Candidate` objects, each O(1) + path string of length O(d)
- No tree data duplication

The `VALUES` expression performs deep copy only for matching results into the result array—after filtering, not during traversal.

## 3.5 Performance Evaluation

This section presents empirical evaluation of the PATHS and VALUES primitives, comparing their performance against traditional imperative loops. The benchmarks use the same dataset, query, and test environment described in the TQuery evaluation (Section 2), enabling direct comparison. Since PATHS and VALUES share the same navigation and filtering implementation—differing only in whether they return path references or deep-copied values—we benchmark VALUES as representative of both primitives. TQuery is not included in these benchmarks because, as demonstrated in Section 2, it exhausts available memory and terminates under concurrent load, making comparison at these concurrency levels unfeasible.

### 3.5.1 Implementations Compared

The benchmark compares the imperative nested for-loop implementation from Section 2 against an equivalent VALUES query:

```
data -> global.data;
results << values
data.companies[*].company.departments[*].teams[*].projects[*]
    where $.status == request.status && $.technologies[*] ==
request.technology
```

### 3.5.2 Benchmark Execution

Both implementations handle concurrent HTTP requests at increasing load levels (100–500 parallel requests). Each concurrency level is repeated 20 times; results report mean and standard deviation. We measure P50 and P95 latency—the response times below which 50% and 95% of requests complete, respectively—as well as peak heap usage and young generation garbage collection frequency via `jstat`. Both latency plots share the same Y-axis scale to allow direct visual comparison.

### 3.5.3 Results

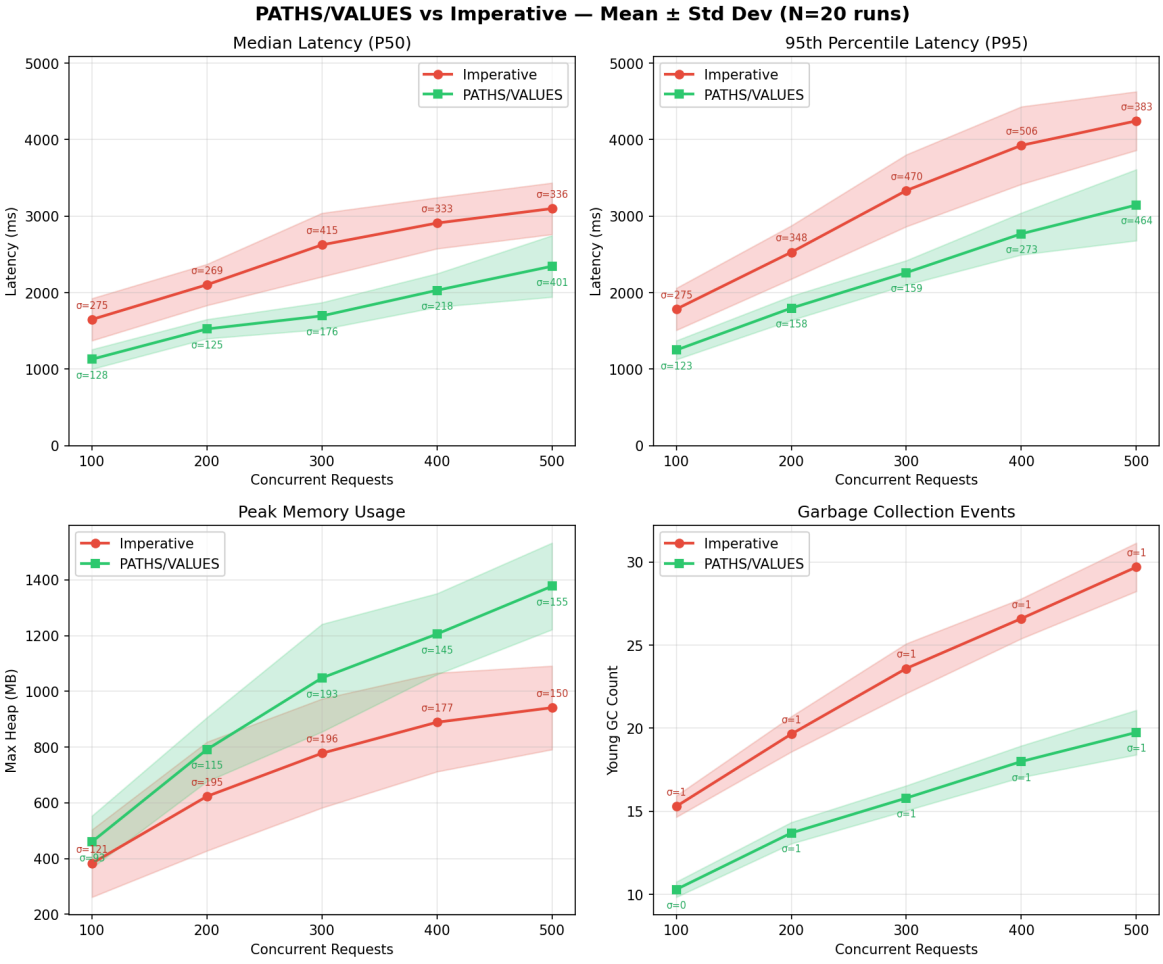


Figure 3: Performance comparison: PATHS/VALUES vs Imperative across concurrency levels (mean ± std dev, N=20 runs)

VALUES achieves **24–35% lower median latency** across all concurrency levels. At 500 concurrent requests, mean P50 drops from 3,099ms to 2,346ms. These gains stem from iterative in-place descent over Jolie's internal structures, early termination via existential

semantics, and reduced instruction overhead—the Jolie-level loop machinery is replaced by Java-native traversal in the runtime.

Garbage collection events decrease by **30–34%** consistently across all levels, indicating lower object allocation during query execution.

Peak memory is the only metric where imperative loops outperform VALUES: the imperative approach uses less heap across all concurrency levels tested, with the gap widening under higher load. This is expected, as VALUES builds intermediate candidate lists during traversal, while the imperative approach processes elements incrementally.

Overall, PATHS/VALUES delivers both simpler code and better latency—contradicting the common assumption that declarative abstractions necessarily sacrifice efficiency for expressiveness.

# Chapter 4

# The Native `path` Type and `pval`

This chapter introduces the native path primitive type and the `pval()` function operator for dereferencing path values. We begin by examining the design rationale for `pval()` as a distinct operator rather than extending existing syntax. The path type is then presented as a first-class primitive, with discussion of its protocol-agnostic communication properties including wire format representation and type validation at message boundaries. The `pval()` function is explored in detail, demonstrating its use for reading values, writing values, and performing deep copies. We examine the two-phase query pattern enabled by separating path collection from value access, and conclude with runtime type checking mechanisms that ensure type safety without repeated validation.

The `paths` expression returns values representing locations in a tree structure. However, these values cannot be used directly—they are symbolic references, not the actual data. Without a mechanism to dereference them, `paths` results would be useless. The `pval()` function bridges this gap, converting path references into accessible values.

## 4.1 Design Consideration: Why `pval()`?

Jolie already supports dynamic field names through parenthesized expressions in paths:

```
fieldName = "status";
x = order.(fieldName);  // Equivalent to order.status
```

A natural extension would be to use similar syntax for path dereferencing:

```
// Hypothetical syntax (NOT implemented)
result = (pathVar).field;
```

However, this approach would break the parser. Jolie's grammar allows parentheses in multiple contexts: grouping expressions `(a + b)`, function calls `foo()`, and dynamic field access `a.(expr).c`. Adding `(expr).field` as a standalone construct would make the grammar ambiguous—the parser could not distinguish between a parenthesized expression and a path dereference without unbounded lookahead, violating context-free grammar requirements.

The `pval()` function provides an unambiguous alternative. The `pval` keyword signals path evaluation, and parentheses serve their standard role of delimiting the argument:

```
result = pval(pathVar).field;  // Unambiguous: evaluate path, then
access field
```

## 4.2 The path Primitive Type

The `path` type is a new Jolie primitive that stores a symbolic location such as `mydata[1]` or `tree.field[2].subfield`. Unlike strings, paths are structured references that the runtime can resolve to actual tree locations.

The implementation adds `ValuePath` to the Jolie runtime:

```
public class ValuePath {
    private final String path;

    public ValuePath(String path) {
        this.path = path;
    }

    public String getPath() {
        return path;
    }
}
```

The `Value` class gains methods for path handling: `isPath()`, `pathValue()`, `setValue(ValuePath)`, and `create(ValuePath)`. Type checking works as expected:

```
res << paths data[*] where $ > 100;
if (res.results[0] instanceof path) {
    println@Console("Got a path value")()
}
```

### 4.2.1 Protocol-Agnostic Communication

As a primitive type, `path` must be sendable as messages across services, just like other primitive types such as `int`, `string`, or `bool`.

```
type PathRequest {
    target: path
}

type PathResponse {
    resolved: path
    value: any
}

interface PathResolver {
    RequestResponse:
        resolve(PathRequest)(PathResponse)
}
```

#### 4.2.1.1 Wire Format: No Type Metadata

Jolie's type system operates on interface contracts, not wire-level metadata. When a `path` value is transmitted, it is serialized as a plain string without special type markers. For example, the path `data[1].field` appears on the wire as:

- **JSON**: `{"target": "data[1].field"}`
- **XML**: `<target>data[1].field</target>`
- **Binary (SODEP)**: String type header + encoded path string

This design keeps wire formats simple and protocol-agnostic. The interface contract—not the message payload—declares that `target` has type `path`.

#### 4.2.1.2 Type Validation at Message Boundaries

Since wire formats carry no type metadata, incoming messages must be validated against their interface contracts. When a service receives a message declaring a `path` field, the type system validates that the received string is a valid instance of the path grammar.

The validation process:

1. Check if the value is already a `ValuePath` object (values from PATHS primitives or already validated within this runtime)
2. If not, attempt to parse the string as a path expression
3. Verify the string conforms to the path grammar (identifiers, field accesses, array indices)
4. Reject malformed paths with a type error

This approach ensures type safety without trusting external input. The interface contract defines what type is expected (`path`), and the type system enforces this by validating the structural correctness of received strings against the grammar.

**Important**: Path validation occurs only at *external* service boundaries—when path values cross input/output ports between independent services. Embedded services operating within the same runtime do not trigger validation. Values already validated—whether from PATHS primitives or previous type checks within the same runtime—skip re-validation for efficiency, but any string received over the wire is always validated to ensure it represents a legitimate path expression.

## 4.3 The `pval()` Function

The `pval()` function—short for "path evaluation"—dereferences a path to access the actual data at that location. Importantly, **`pval()` works both as an lvalue (for writing/modifying data) and as an rvalue (for reading data)**, providing complete bidirectional access through path references.

### 4.3.1 Syntax

```
pval( path-clause ) suffix*
```

The function takes a path value and returns a reference to the actual data. Optional suffixes (`.field`, `[n]`) allow further navigation from the dereferenced location.

### 4.3.2 Reading Values

```
mydata[0] = 100;
mydata[1] = 200;
mydata[1].child = "one";
mydata[2] = 300;
mydata[2].child = "two";

// Get paths to elements > 150
res << paths mydata[*] where $ > 150;
// res.results contains: mydata[1], mydata[2]

// Dereference to get actual values
readVal = pval(res.results[0]);        // Returns 200
childVal = pval(res.results[0]).child; // Returns "one"
```

Path navigation composes after dereferencing:

```
mydata[1].children[0] = "first";
mydata[1].children[1] = "second";
arrayVal = pval(res.results[0]).children[1];  // Returns "second"

mydata[1].nested.items[1].name = "item1";
nestedVal = pval(res.results[0]).nested.items[1].name;  // Returns
"item1"
```

### 4.3.3 Writing Values

The `pval()` function also works as an lvalue—the left side of an assignment:

```
toUpdate << paths orders[*] where $.status == "pending";

for (i = 0, i < #toUpdate.results, i++) {
    pval(toUpdate.results[i]).status = "processed"
}
```

The parser recognizes `pval(...) = expr` as an assignment statement, generating a `PvalAssignStatement` AST node.

### 4.3.4 Deep Copy

For tree assignment (the `<<` operator), `pval()` supports deep copying entire subtrees into path-referenced locations:

```
// Create a template with nested structure
newData << {
    x = 100
    y = 200
    label = "point"
    nested.a = 1
    nested.b = 2
};

// Find matching records and copy template into each
res << paths data._[*] where $.status == "pending";
for (i = 0, i < #res.results, i++) {
    pval(res.results[i]).extra << newData
}
// Each matching record now has an 'extra' subtree with the full
template
```

The parser recognizes `pval(...) << expr` as a deep copy statement, generating a `PvalDeepCopyStatement` AST node that performs a full tree copy to the dereferenced location.

#### 4.3.5 Key Capabilities

The `pval()` function provides comprehensive access to path-referenced data:

- **Works as lvalue**: Can modify data through path references (`pval(path).field = value`)
- **Works as rvalue**: Can read data through path references (`x = pval(path).field`)
- **Works with nested fields**: Supports deep navigation (`pval(path).field.subfield`)
- **Works with arrays**: Supports array indexing (`pval(path).array[0]`)
- **Supports both = and <<**: Simple assignment vs deep copy
- **Can be used in expressions**: Arithmetic (`total = pval(p1).value + pval(p2).value`), conditions (`if (pval(path).salary > 50000)`)
- **Path-to-path operations**: Direct transfers (`pval(dest) << pval(src)`)

### 4.4 Reference vs. Copy Semantics

A critical distinction exists between `values` and `paths` with `pval()`:

- **`values`** returns **deep copies**—modifying results does not affect the original data
- **`paths` + `pval()`** provides **references**—modifications through `pval()` affect the original data

```
readFile@File({ filename = "data.json", format = "json" })(data);

// VALUES: deep copy semantics
res << values data._[*] where $.status == "active";
res.results[0].status = "modified";
// data._[0].status is UNCHANGED - res contains independent copies

// PATHS + pval: reference semantics
res << paths data._[*] where $.status == "active";
pval(res.results[0]).status = "modified";
// data._[0].status IS CHANGED - pval provides a reference to original
```

This distinction determines which approach to use:

- Use `values` when you need isolated copies for transformation or output
- Use `paths` + `pval()` when you need to modify the original data structure

## 4.5 Two-Phase Query Pattern

The combination of `paths` and `pval()` enables a two-phase pattern:

1. **Query phase**: Use `paths` to find locations matching criteria
2. **Access phase**: Use `pval()` to read or modify those locations

```
// Phase 1: Find all high-priority items
highPriority << paths tasks[*] where $.priority == "high";

// Phase 2: Process each one
for (i = 0, i < #highPriority.results, i++) {
    path = highPriority.results[i];
    println@Console("Processing: " + pval(path).name)();
    pval(path).processed = true
}
```

The `path` values act as stable references—even as the loop modifies data, the paths remain valid.

## 4.6 Runtime Type Checking

The `pval()` function checks the nominal type of its argument—verifying it is a `path` type value:

```
// PvalHelper.resolveVariablePath()
if (!value.isPath()) {
    throw new FaultException("TypeMismatch",
        "pval requires a path type value, got: " + value.strValue());
}
```

This check occurs at evaluation time:

```
x = "not a path";
y = pval(x);  // Throws: TypeMismatch: pval requires a path type value
```

Crucially, the implementation checks only the nominal type, not the internal path structure. This is sound because `path` values have a strong origin guarantee: they can only be created by the `paths` primitive or received via message passing. Both sources ensure structural validity:

- **PATHS primitive**: Constructs `ValuePath` objects directly from successful tree navigation
- **Message passing**: Type system enforces `path` type contracts at communication boundaries

Once the nominal type check succeeds, the runtime has a guarantee that the internal path string is well-formed—no repeated parsing or validation is needed.

# Chapter 5

# Conclusion

This thesis presented PATHS and VALUES, native Jolie language primitives for declarative tree querying. By extending the Jolie parser, runtime, and type system, these primitives enable expressive queries over tree-structured data without external dependencies or performance penalties.

## 5.1 Summary of Contributions

The main contributions of this work are:

1. **Native path operations**: Six composable operations (`.field`, `.*`, `[n]`, `[*]`, `..field`, `..*`) for flexible tree navigation, integrated directly into the Jolie language syntax.
2. **WHERE clause with existential semantics**: A filtering mechanism using the `$` operator to reference the current value, supporting boolean expressions with existential matching—a condition succeeds if any value satisfies it.
3. **The HAS operator**: Structural filtering based on field existence rather than field values, distinguishing "field absent" from "field present but empty."
4. **Native path type and pval**: A new `path` type representing tree locations, with `pval()` for dereferencing paths to actual values, enabling two-phase query-then-access patterns.
5. **Zero-copy evaluation**: By operating directly on Jolie's internal `Value` and `ValueVector` structures, PATHS and VALUES avoid the deep cloning required by TQuery's specification, eliminating the substantial memory overhead observed in benchmarks.

## 5.2 Addressing the Challenges

Returning to the hierarchical querying challenges illustrated in Chapter 2, PATHS and VALUES provide unified solutions where previous approaches required trade-offs.

**The grandfather name-matching query**, which required imperative nested loops (Listing 1) or TQuery's multi-stage pipelines (Listing 2), reduces to:

```
values data._[*] where
    $.sex == "Male" &&
    $.name == $.children[*].children[*].name
```

The query traverses all persons, filters for males, and checks if any grandchild shares the grandfather's name—all in three lines without explicit loops or intermediate variables.

**The companies-departments-teams-projects filtering**, which necessitated MongoDB's complex nested `$reduce` or memory-intensive `$unwind` cascades, becomes:

```
values data.companies[*].company.departments[*].teams[*].projects[*]
    where $.status == "in_progress" && $.technologies[*] == "Python"
```

The path expression navigates four levels deep with wildcard array expansion, while the WHERE clause filters by status and technology in a single declarative statement—eliminating the need for nested `$reduce` functions or memory-intensive `$unwind` stages.

Where previous approaches forced choices between expressiveness and efficiency, schema design and query complexity, or memory overhead and readability, PATHS and VALUES achieve all objectives simultaneously through native integration with Jolie's tree-structured data model.

## 5.3 Future Work

Several directions remain for future exploration:

**Additional operators**: The current WHERE clause supports comparison and logical operators. Extending with aggregation functions (`count`, `sum`, `avg`) or set operations (`in`, `all`) would increase expressiveness. A `project` operator, inspired by TQuery, could reshape query results by selecting and renaming specific fields, eliminating manual iteration for result formatting.

**Query optimization**: The current implementation evaluates queries eagerly. Lazy evaluation or query plan optimization could improve performance for large datasets or complex conditions.

**Integration with type system**: Stronger static guarantees about path validity and result types could catch errors at compile time rather than runtime.

**Formal semantics**: A formal specification of PATHS and VALUES semantics would enable correctness proofs and guide future extensions.

**Path manipulation service**: A dedicated service for `path` value manipulation would enable operations like path concatenation, decomposition, and conversion between path values and strings, supporting dynamic path construction and introspection.

# Bibliography

[1] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with Jolie," *Web Services Foundations*, pp. 81–107, 2014, [Online]. Available: https://www.fabriziomontesi.com/files/mgz14.pdf

[2] R. Pike, "Simplicity is Complicated." [Online]. Available: https://www.youtube.com/watch?v=rFejpH_tAHM

[3] S. Giallorenzo, F. Montesi, L. Safina, and S. P. Zingaro, "Ephemeral data handling in microservices with Tquery," *PeerJ Comput. Sci.*, vol. 8, p. e1037, 2022, doi: 10.7717/peerj-cs.1037.