RedType

A Type Safe Query Language

Emil Gade, Frederik Brandi Nielsen, Kasper Vestergaard Jensen,
Magnus Storgaard, Martin P. Tielemans, Raman Dara Aziz

Computer Science, cs-25-dat-4-02-, 2025-05

P4 Project

Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
RedType

Theme:
Design, Definition and Implementation of a programming language

Project Period:
Spring Semester 2025

Project Group:
Group 2

Participant(s):
Emil Gade
Frederik Brandi Nielsen
Kasper Vestergaard Jensen
Magnus Storgaard
Martin P. Tielemans
Raman Dara Aziz

Supervisor(s):
Brian Nielsen

Copies: 1

Page Numbers: 149

Date of Completion:
April 13, 2026

Abstract:

RedType is an alternative to Redis with lua. It is a type safe in-memory database system addressing Redis' lack of native type safety and schema enforcement. It introduces a statically typed schema, a C-family inspired query language, atomic multi-threaded operations, and robust Option types. Implemented in Rust with LALRPOP and Tokio, RedType employs pessimistic locking for concurrency. Testing confirmed must-have requirements, demonstrating a balance between type safety and performance. While further enhancements are planned, RedType provides a foundation for a reliable, type safe key-value store where data integrity is critical.

Contents

Preface

Aalborg University, April 13, 2026

Emil Gade
<egade23@student.aau.dk>

Frederik Brandi Nielsen
<fbni23@student.aau.dk>

Kasper Vestergaard Jensen
<kj23@student.aau.dk>

Magnus Storgaard
<mstorg23@student.aau.dk>

Martin P. Tielemans
<mtiele23@student.aau.dk>

Raman Dara Aziz
<raziz23@student.aau.dk>

This document was typeset using LaTeX with the article document class. Figures were created using Excalidraw. Code development was done using Rust for the server implementation and TypeScript for the client. Version control was managed through Git, and the compiler framework utilized LALRPOP for parsing. AI tools like GitHub Copilot and ChatGPT were used for code assistance, error correction, and documentation improvements.

1 Introduction

Redis is an in-memory key-value store renowned for its simplicity and high performance, making it a popular choice across a wide range of industries. It supports fast data access and flexible data structures. However, Redis lacks built-in type safety, since all data is stored as untyped binary-safe strings, which can lead to confusion, inconsistent behavior, and runtime errors, particularly in complex or collaborative systems where developers make conflicting assumptions about the data.

To address these limitations, RedType has been developed as a Redis-inspired system with native support for type safety. It introduces explicit schema definitions and a statically typed scripting environment, with type checking enforced before query interpretation. These features reduce the likelihood of runtime errors and enforce consistency across teams. In contrast to Redis's single-threaded architecture, RedType also supports multithreaded, asynchronous execution, enabling safe and efficient concurrent query processing.

This report begins by analyzing Redis's functionality and its limitations, then outlines the motivation, philosophy, design, and implementation of RedType. Although RedType does not match Redis in raw performance, it offers improved safety, predictability, and developer confidence. Through its statically typed schema system and dedicated query language, RedType enables more reliable data manipulation. By integrating formal type enforcement with atomic, multithreaded execution, RedType provides a powerful and safe alternative for high-performance key-value storage.

2 Problem Analysis

This section provides an analysis of Redis. The analysis begins by examining Redis's core features, architecture, and industrial applications to understand its widespread adoption and strengths. Following this, the limitations and challenges associated with Redis's lack of type safety are explored. The problem analysis demonstrates how these limitations can lead to runtime errors and inefficiencies in distributed development environments. Finally, existing solutions that attempt to address these challenges are evaluated, highlighting their shortcomings and paving the way for the proposed solution.

2.1 Redis

Redis, short for Remote Dictionary Server, is a high-performance, in-memory key-value store widely used for applications like real-time data processing and message brokering. As an open-source NoSQL system, e.g a relation-less database system, Redis distinguishes itself by storing data directly in the system's primary memory (RAM) as opposed to traditional databases that rely on secondary memory (e.g Harddrives and SSD's). This in-memory data storage approach enables Redis to deliver significantly faster response times, while still supporting features such as atomic operations, real-time event handling, and scalable distributed architectures. [45] Of these features atomicity is the most important, since it allows operations to be executed as indivisible units without any interference from others. [43] This means all operations either complete in their entirety or not at all, with this being essential in concurrent systems where multiple processes may attempt to access or modify shared resources simultaneously. [49]

At its core, Redis utilizes a key-value storage model where each pair consists of a unique key and its corresponding value, eliminating the need for complex schema and table joins. For instance, in the pair `{"age":22}`, "age" serves as the key, and the integer `22` is its associated value. Data retrieval is highly efficient as it involves directly referencing the unique key. When a key is queried, Redis employs a hashing algorithm to determine the memory address where the value is stored. This mechanism typically results in an average-case time complexity of $O(1)$ for both read and write operations. Unlike more complex relational databases that might necessitate multiple table joins or intricate queries to access data, Redis retrieves data directly via its key. [34] [21]

Redis can be employed in a variety of different environments, and often in union with the scripting language Lua: Publish/Subscriber (pub/sub) messaging, caching, and even as a main database. In the following, Figure 1 illustrates a use case scenario for Redis as an intermediary data layer in a backend architecture.

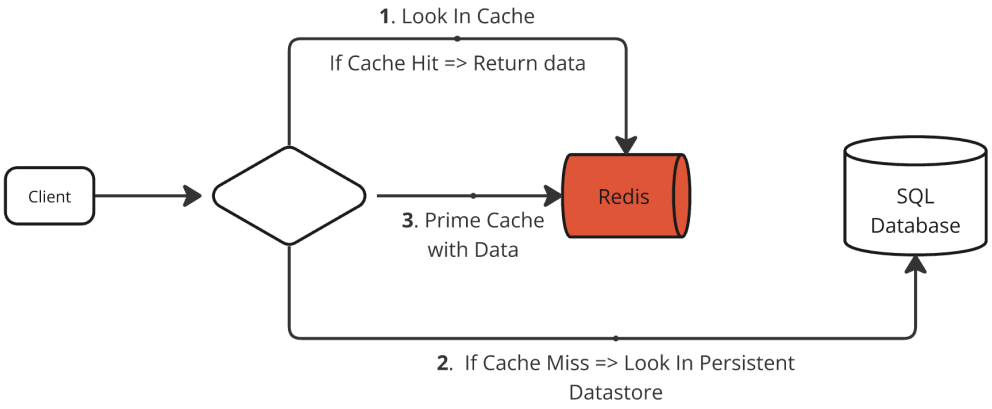

Figure 1: Example use of Redis. Based on source. [8]

Figure 1 illustrates how a client interacts with a Redis instance through a backend application. In this case, Redis acts as a high-speed intermediary, efficiently handling requests by storing and retrieving structured data in memory. This approach enables applications to process real-time events, manage temporary state, and optimize performance. Applicatons such as real-time analytics, leaderboards in video games [36], and queue systems all benefit from Redis' utilization of in-memory storage. [15]

Another way to use Redis is as a primary database, particularly when performance and low latency are crucial components. While Redis often is associated with caching layers or supplementary storage, its support for data persistence through RDB snapshots, and Append-Only File (AOF) logs has significantly increased. RDB snapshots periodically save the dataset to disk but can risk losing recent data, while AOF logs record each write operation for greater durability at the cost of higher write overhead. [15]

2.2 Use Cases of Redis

Redis has gained significant adoption across various use cases due to its versatility and performance characteristics. This section explores how Redis can be leveraged to solve specific challenges.

Black Friday Example

During high-traffic sales events such as Black Friday, e-commerce platforms must efficiently manage inventory updates at scale. Redis is well-suited for this context due to its in memory data store, which supports low-latency read and write operations. It enables rapid stock lookups and updates, which is essential when thousands of users are actively browsing and purchasing items. In a system, product inventory can be stored using simple key-value pairs. Redis provides commands like GET and SET to retrieve or assign values, which can be used to implement stock reservation and purchase flows. [31]

The following listing shows how Redis can be used to reserve one unit of a product with a given ID by incrementing the reserved stock counter. This is written in JavaScript using a Redis library:

```
import { createClient } from 'redis';

const client = createClient();
await client.connect();

const productKey = "product:bf_special_item_001";

// Check availability
const available = parseInt(await client.get(`${productKey}
:stockAvailable`));
const reserved = parseInt(await client.get(`${productKey}:stockReserved`));
const effectiveStock = available - reserved;

if (effectiveStock >= 1) {
    // Reserve stock
    await client.incrBy(`${productKey}:stockReserved`, 1);
    console.log("SUCCESS: Reserved item.");
} else {
    console.log("FAILURE: Not enough stock.");
}
```

Listing 1: Redis example: Reserving a product during Black Friday

Once an item has been successfully reserved and the customer completes payment, the system must confirm the purchase by reducing both the available and reserved stock. This step extends the earlier logic and finalizes the inventory update:

```
// Continuing from the previous example

const quantityPurchased = 1;

if (
    quantityPurchased > 0 &&
    quantityPurchased <= reserved &&
    quantityPurchased <= available
) {
    await client.decrBy(${productKey}:stockReserved, quantityPurchased);
    await client.decrBy(${productKey}:stockAvailable, quantityPurchased);
    console.log("SUCCESS: Purchase confirmed and stock updated.");
} else {
    console.log("FAILURE: Invalid quantity or insufficient stock.");
}
```

Listing 2: Redis example: Confirming a purchase after reservation

This example finalizes the transaction by adjusting inventory counters for both available and reserved quantities. While these example simplify the overall process, it demonstrates a typical use case for Redis in a high-traffic environment. In these examples, there is no null checks or type validations. [31]

Financial Services

Financial institutions leverage Redis for time-sensitive operations requiring both speed and reliability. Trading platforms use Redis to store market data feeds, enabling real-time analytics and algorithmic trading decisions. Banks implement Redis for fraud detection systems, where pattern matching against suspicious activities must occur in milliseconds. Payment processors rely on Redis for transaction processing and rate limiting to prevent system overload. Additionally, Redis serves as a distributed lock manager for financial transactions, ensuring consistency across distributed systems. [35]

Other sectors such as healthcare and gaming can also leverage Redis for use a variety of different use cases. The healthcare industry might use it for patient data caching and real-time monitoring, while the gaming industry can use it for session management, leaderboards, and content recommendation. These industries benefit from Redis's low latency and scalable architecture. [30, 36] Even though Redis excels in speed and flexibility, its approach to data management presents challenges. It supports various data structures,

but the lack of strict type enforcement can lead to inconsistencies and unexpected behavior in complex applications. Since Redis is increasingly used in more critical systems, understanding these limitations is essential.

2.3 The Issue with Data Types in Redis

While Redis offers outstanding performance, its approach to data types presents significant drawbacks. A primary limitation is its lack of strong data typing, potentially leading to complications in large-scale and complex applications. Without strict type enforcement, developers risk encountering issues and bugs stemming from mismanaged data types.

Redis currently supports a variety of flexible data structures, but treats all of them as binary-safe strings, meaning Redis does not inherently distinguish between data types like integers, booleans, or floats. Core data structures in Redis include strings, lists, sets, sorted sets, hashes, bitmaps, geospatial indexes, and streams. Each structure is designed for specific use cases, such as storing ordered collections (lists), unique items (sets), or key-value pairs (hashes). Redis provides commands to manage these structures, enabling operations like adding, removing, and manipulating data. [29]

However, the absence of explicit type enforcement introduces challenges, particularly for operations involving different data types. For example, Redis stores numeric values as strings, necessitating manual conversion. This can result in unexpected behavior when performing arithmetic operations directly on retrieved values without conversion. Some languages such as JavaScript might interpret the operation as string concatenation instead of arithmetic [23]. Similarly, boolean values are represented by strings like `"1"` and `"0"`, complicating logical operations. Consequently, developers must explicitly manage type consistency, potentially causing errors and inefficiencies.

2.4 Type Safety in Distributed Development

In distributed software development teams, assumptions about data structures often lead to runtime errors and inefficiencies. Consider a scenario where two developers, Alice and Bob, are collaborating on a user management system employing Redis as the data storage solution.

Developer Implementations

Alice has implemented a user authentication module in Python that stores user credentials. Her implementation records the user's password hash as a string and a list of permissions:

```python
# Alice's Implementation
def store_user(username, password_hash, permissions):
    # Store password hash as a string
    redis_client.set(f"user:{username}:password", password_hash)

    # Store permissions as a Redis list
    redis_client.rpush(f"user:{username}:permissions", *permissions)

    # Store last login time as Unix timestamp (integer)
    redis_client.set(f"user:{username}:last_login", int(time.time()))
```

Listing 3: Alice's Implementation

Here, Alice uses `.set`, which causes Redis to store both the password hash under the key `user:{username}:password` and the `last_login` timestamp as a Unix timestamp integer. However, Redis treats all stored values as binary-safe strings, meaning they are stored as raw byte sequences without type enforcement. The command, `RPUSH` appends each permission to a Redis list stored at `user:username:permissions`, or creates the list if the list does not exist.

Meanwhile, Bob develops a user statistics module tracking login patterns. His implementation assumes different data types for the same keys:

```python
# Bob's Implementation
def track_login_activity(username):
    # Increment login counter
    redis_client.incr(f"user:{username}:last_login")

    # Retrieve permissions as a string
    login_history = redis_client.get(f"user:{username}:permissions")
    if login_history:
        # Attempt to analyze permission usage patterns
        analyze_permissions(login_history)
```

Listing 4: Bob's Implementation

Bob assumes that `last_login` is the amount of times a user has logged in and therefore uses `redis_client.incr` to increment `last_login`, which only works correctly on integer values. When using `INCR`, Redis will increment the value even though it is stored as a binary-safe string. This is because Redis implicitly treats string values that represent valid

integers as numeric. However due to Alice's implementation, there occurs some errors. He also assumes that permissions are stored in a string and therefore uses `redis_client.get` to retrieve permissions as a string, which only works on string types.

The Resulting Confusion

When these modules are integrated, several runtime errors occur:

1. Bob's code attempts to increment `last_login`, which Alice set as a timestamp. While this operation technically succeeds (since Redis interprets both as strings), it produces semantically incorrect data.
2. Bob's code attempts to read `permissions` as a single string with `get`, but Alice stored it as a list with `rpush`, resulting in a type mismatch error:

```
redis> GET user:alice:permissions
(error) WRONGTYPE Operation against a key holding the wrong
kind of value
```

Errors such as these are discovered only during runtime testing, causing delays and requiring code refactoring.

A schema-based type safety approach would prevent these issues by establishing a contract for key-value data types. With type checking middleware, Bob's attempt to increment a timestamp or retrieve a list as a string would fail during development rather than runtime. For readers who wish to experience this issue firsthand, a demonstration is available in the attached files within "redis-type-mismatch-demo.zip", which includes a `README` for guidance.

2.5 Schema Design in Other Languages

Schemas can play a crucial role when defining the structure or constraints of data. Incorporating schemas into a language for data handling offers numerous benefits, including type safety and data integrity. [24]

Schemas offer several key advantages when working with data. One of the most obvious benefits is type safety. Schemas allow data types to be explicitly defined for different fields, ensuring that invalid or incompatible data cannot be stored. This contributes directly to data integrity, as schemas enforce constraints that guarantee data conforms to the required types and expected values. As a result, inconsistencies and errors are minimized, leading to more reliable datasets. Schemas also provide a clear and concise way to document data structures, making it easier for developers to work with. For example, if a backend team

has prematurely defined the structure of certain data, the frontend team can mock the data without needing real data from the backend. [24] To illustrate some of these benefits, an example have been made in SQL.

SQL Example: When using SQL, developers define the structure of the data using a declarative schema [9]. SQL schemas are made using the ´CREATE TABLE´ statement, which enables type enforcement through definitions and constraints:

```
1    CREATE TABLE users (
2    id INT PRIMARY KEY,
3    name VARCHAR(255),
4    email VARCHAR(255) UNIQUE,
5    created_at TIMESTAMP
6 );
```

Listing 5: SQL CREATE TABLE Example

In the example above in Listing 5, the `CREATE TABLE` statement defines a `users` table with columns `id` (integer and primary key), `name` (string), `email` (string and unique), and `created_at` (timestamp). This schema enforces these data types and constraints, thereby ensuring integrity and type safety.

In contrast to SQL, which uses schemas for type safety and data integrity, Redis lacks this built-in enforcement. As shown earlier, this absence can lead to runtime errors and inconsistencies. Adding a robust schema layer to a Redis-like system offers a direct solution to these fundamental type-related problems.

2.6 Existing Solutions

To further address the limits of Redis, both in terms of type safety and asynchronous performance, various solutions have been created. This section examines existing solutions, their benefits, and shortcomings.

Redis OM

Redis OM is an object-relational mapping, which abstracts Redis operations into object-oriented models, allowing developers to interact with Redis using classes and objects. In Redis OM schemas are enforced and validated at runtime, but is also mapped to objects. However, it transforms Redis into a form of relational database, which does not fit all use cases. Furthermore, the added type safety is at runtime on the client and not native. This can result in already mentioned errors, such as direct access commands and performance overhead. In addition, Redis OM fails to handle complex sequences of operations that are treated as a single, indivisible unit of work (transactions) without blocking the event loop. [37]

DragonflyDB

DragonflyDB is an multithreaded in-memory database that is aimed at solving Redis' issues with regard to singlethreaded execution. It furthermore boasts an increase in performance, due to being vertically scalable and allowing for parallel command execution, among a large set of optimizations. Since DragonflyDB is multithreaded they have to enforce atomic transactions in a different way than Redis' halting of the event loop. They use shard-locking, which is locking of individual shards (a smaller section of keys) during a transaction. Multiple shards could also be locked in case of interactions between data that crosses more than one shard, resulting in atomicity being upheld. [7] [6]

However, while DragonflyDB solves performance and singlethreaded issues, it still retains Redis' lack of types. Resulting in previously mentioned issues such as runtime and type safety errors.

2.7 Exploring the Potential for a Strongly-Typed Redis System

Redis OM attempt to add type safety onto Redis, often as runtime validation, but their approaches are fundamentally limited because this safety is not native. Since the Redis instance itself remains unaware of external schemas, validation can be bypassed by raw commands or other clients, directly leading to data inconsistency. Treating type safety as an add-on, rather than an integral database feature, fails to guarantee data integrity or leverage potential server-side optimizations. These solutions may also be affected by Redis' inherent singlethreaded architecture. As a multithreaded alternative, DragonflyDB addresses this particular performance concern, but continues to operate without native type safety. Current approaches to add type safety could be viewed as separate components addressing different aspects of the problem rather than comprehensive solutions.

These existing solutions approach type safety as a separate component each targeting individual aspects of the problem rather than addressing the problem at its core. Given the limitations of existing solutions, we consider three primary options for developing a more comprehensive type-safe key-value store with type:

1. **Build on Redis Commands**: Create a type safe layer that compiles to native Redis commands while enforcing schemas and types.
2. **Utilize Lua Scripting**: Leverage Redis' Lua scripting capabilities to implement type checking and complex operations server-side.
3. **Create a Redis Alternative**: Develop a new Redis-inspired database system with native type safety built into its core design.

Each approach offers different trade-offs between compatibility, performance, and implementation complexity. The following section will evaluate these options in detail to deter-

mine which approach best addresses type safety challenges while maintaining the performance benefits that make Redis valuable. This evaluation will lead to the introduction of our proposed solution, a strongly-typed Redis system, hereby referred to as *RedType*.

2.8 Evaluation of Implementation Approaches

In section 2.7, three different potential approaches for developing a type safe Redis-like system were identified in a brainstorm: building on native Redis commands, utilizing Lua scripting capabilities, or creating a Redis-inspired alternative with built-in type safety. This section evaluates each option in detail and provides justification for our final choice.

Option 1: Redis Commands

This approach involves using native Redis commands (e.g., `GET`, `SET`) for data interaction. While offering simplicity and direct compatibility with Redis features and no inherent execution time limits, it can become cumbersome for expressing complex, multi-step logic. Although Redis provides mechanisms like transactions (`MULTI/EXEC`) and optimistic locking (`WATCH`) to ensure atomicity, these tools require careful orchestration by the client and offer limited support for conditional or rollback logic in case any errors occur, allowing the database to discard any unwanted changes. As a result, implementing advanced query semantics typically demands many discrete commands and careful coordination, increasing the risk of race conditions and introducing nontrivial network overhead due to multiple client-server round-trips [40]. These challenges make native Redis commands a less ideal foundation for building a high-level, type safe query language that requires expressive and atomic composition of operations.

Option 2: Lua Scripts

Redis provides support for server-side Lua scripting, which appears to offer a solution for executing complex operations atomically. However, our investigation revealed several critical limitations that make Lua scripts unsuitable for implementing a robust type safe query language:

1. **Global Blocking**: Lua scripts block Redis' singlethreaded event loop during execution, preventing other client operations from being processed. As script complexity increases, this creates significant performance bottlenecks in high-throughput environments [32].

2. **Execution Time Limits**: Redis imposes strict execution time limits on Lua scripts (default 5 seconds) to prevent indefinite blocking. Complex queries involving large datasets could easily exceed this limit, leading to script termination and incomplete operations [33].

3. **No Transaction Rollbacks**: While Lua scripts execute atomically, they lack true transaction capabilities, particularly the ability to roll back changes if errors occur mid-execution. This creates data integrity risks for complex operations [39].

4. **Limited Type Safety**: Lua itself is dynamically typed, making it difficult to enforce the strong type guarantees we need. Any type checking would require complex validation logic within each script [41].

A sample Lua script implementation reveals these limitations:

```
local script = [[

  local results = {}
  local keys = redis.call('KEYS', 'order:*')

  for i, key in ipairs(keys) do
    local status = redis.call('HGET', key, 'status')
    local date = redis.call('HGET', key, 'orderDate')

    if status == 'shipped' and date > '2023-01-01' then
      local id = redis.call('HGET', key, 'id')
      local total = redis.call('HGET', key, 'total')
      local name = redis.call('HGET', key, 'customerName')

      table.insert(results, {
        id = id,
        total = total,
        customerName = name
      })
    end
  end

  return cjson.encode(results)
]]

EVAL script 0
```

Listing 6: Lua Script with Inherent Limitations

This script highlights key limitations of using Lua in Redis for complex queries. The script's multiple `HGET` calls per key increase execution time, risking termination if the default time limit is exceeded. Additionally, Lua offers no rollback on errors. If a failure occurs mid-script, partial changes remain. Finally, Lua's dynamic typing and Redis's string-based storage make it difficult to enforce strong type guarantees, requiring manual validation.

Option 3: Building a Redis with Lua Alternative

After thoroughly evaluating the limitations of both direct Redis commands and Lua scripting, it becomes evident that neither approach fully satisfies the requirements for a type safe, performant query language. These limitations necessitate a different solution: rather than building on top of Redis with its inherent constraints, the project aims to create a Redis-inspired alternative database system that natively supports:

- **Native Type Safety**: Built-in schema validation and type checking at the storage level, eliminating the need for external validation layers.
- **Queries with Rollbacks**: Support for complex operations that can be safely rolled back if errors occur, ensuring data integrity.
- **Non-Blocking Query Execution**: Multithreaded architecture that allows complex queries to execute without blocking other operations, using fine-grained locking instead of global locks.
- **Unlimited Execution Time**: No arbitrary timeout constraints for complex operations, enabling comprehensive data analysis queries.

Building a custom system addresses the fundamental limitations identified in the analysis of Redis and creates a solution that meets the needs for type safety in high-performance data operations.

Final Decision

After carefully evaluating all three options, we have decided to pursue Option 3: Building a Redis Alternative. Option 1 (Redis Commands) would require complex client-side logic and multiple network round-trips for any non-trivial operations, introducing both performance overhead and potential race conditions. Option 2 (Lua Scripts) offers better atomicity but introduces global blocking, execution time limits, and still lacks native type safety guarantees. Only Option 3 would address the core issues at the architectural level, providing a solution that is truly type-safe by design rather than through additional validation layers.

The system will therefore be implemented as a standalone database that maintains Redis's performance advantages while adding the critical features needed for robust type enforcement and advanced querying capabilities.

3 Problem Definition

The previous chapter explained Redis, its use cases, limitations, and alternative solutions developed to address these challenges.

As explained, Redis is widely used as a for its key-value store, that is used in various industries due to its high speed achieved by storing data directly in memory.

However, despite its efficiency and performance, Redis has several shortcomings that introduce challenges as applications grow in scale and complexity. Redis lacks built-in type safety, which can lead to type related runtime errors, as demonstrated in 2.4. These errors are only discovered during runtime testing, which negatively impacts development. Redis also stores all data as binary-safe strings, requiring manual type conversion and leading to potential runtime errors and unexpected behavior, as described in 2.3.

To solve these issues Redis OM has been developed in an attempt to improve Redis's capabilities. However, it relies on runtime validation rather than native schema enforcement, and thus fall short of fully addressing the type safety problem. DragonflyDB's multithreaded implementation offers improved performance and concurrency over Redis' singlethreaded model. Its success in this area inspires RedType to adopt similar multithreading capabilities. However, DragonflyDB itself does not inherently solve schema enforcement or type safety issues, leaving developers to manage these concerns at the application layer.

In 2.7, different options for addressing these issues were explored. Based on the limitations of Redis, the trade-offs of existing solutions, and the potential of designing a new programming language, the following problem statement is proposed:

3.1 Problem Statement

How can an in-memory database system inspired by Redis be designed with a schema- and query language that enforces type safety before runtime, executes complex queries atomically, supports multi-threaded execution while still preserving the performance benefits of in-memory storage?

4 RedType Philosophy

This section outlines the core philosophy guiding the development of RedType, ensuring a shared vision and consistent direction among all team members.

4.1 Improving on Redis: Type safety and Concurrency

RedType was created to address certain drawbacks found in Redis. The primary goals are to introduce type safety, and to overcome limitations caused by its singlethreaded design. RedType utilizes commands, similar to those in Redis, and will also incorporate the RedType Query Language, which serves a purpose comparable to Lua scripting in Redis. This allows developers familiar with Redis to adopt a similar approach while gaining the advantage of type safety provided by the RedType Query Language. The language design aims to ensure clarity regarding data types, thus reducing potential confusion, particularly within larger development teams.

4.2 Database Schema as a Contract for Typesafety

To effectively enforce type safety, RedType mandates the use of an explicit schema definition. This schema serves as a formal contract between the application developer and the database instance. It precisely defines the expected data types (like `String`, `Int`, `Double`, `Boolean`), field names, and basic constraints for the data being managed. By establishing this clear contract upfront, the schema becomes the authoritative source of truth for data structure, which the RedType server then enforces during all operations. This explicit definition is crucial for preventing type ambiguity and potential runtime errors common in schema-less systems.

4.3 Why the RedType Query Language is Necessary

Consider scenarios requiring consecutive data operations, where fetching specific data depends on the results obtained from preceding queries. Performing such tasks with individual commands can lead to significant back-and-forth communication between the client and the server. If the client and server are geographically distant, the latency introduced by this communication pattern can become substantial. For instance, executing ten dependent queries consecutively, each incurring 100ms of round-trip time, would result in a total delay of one second.

The RedType Query Language is designed to alleviate this issue. It enables the execution of complex logic directly on the RedType database instance. This allows for creating, reading, updating, and aggregating data within a single, unified query operation greatly

enhancing the efficiency of tasks involving multiple dependent steps or complex data aggregation requirements.

4.4 Query Language Syntax: Prioritizing Familiarity

While Redis effectively utilizes Lua for scripting, RedType deliberately diverges in its approach to language syntax. Instead of adopting a Lua like structure RedType incorporates a RedType Program Language with a syntax influenced by the C family including languages such as C++, Java, C#, and JavaScript. This design choice is rooted in the philosophy of maximizing developer familiarity and minimizing potential sources of friction often encountered when switching between programming paradigms.

Several characteristics of Lua, though effective within its own ecosystem, were identified as potential hurdles for developers accustomed to more mainstream syntaxes. Notably Lua uses tables as its sole data structuring mechanism. These tables employ one based indexing when used as sequences. For example list style initialization in table constructors starts indices at one. The length operator which is denoted by the number sign character is also defined accordingly for these sequences, contrasting with the zero based indexing standard in most C style languages. This difference can frequently lead to off by one errors and requires conscious adaptation. Furthermore Lua's use of keywords like `end` to delineate code blocks is evident in control structures and `do end` blocks. This use differs from the widespread convention of using curly braces, which familiar to a vast number of developers. Another significant factor was Lua's default global scope for variables. This characteristic requires explicit use of the `local` keyword to define variables with lexical scope. Using `local` helps avoid potential namespace collisions and unintended side effects particularly in larger or collaborative projects. [13]

Opting for a C like syntax makes the RedType Query Language more intuitive and predictable for many developers. This leverages widespread knowledge shortening the learning curve and keeping the focus on application logic. Such familiarity enhances the developer experience and promotes clarity directly supporting RedType's goals in its type safe environment.

4.5 Core Language Principles: Typing, Name Resolution, and Scope

Beyond its C-family inspired syntax, the RedType Language is based on several core principles for handling data types, resolving names, and determining their visibility. These choices are important for achieving RedType's goals of type safety, code clarity, and developer predictability.

RedType will use static typing, meaning the type of every variable and expression is statically checked before the program is run. This also makes code clearer, as types explicitly

show the intended use of data. This supports RedType's goal of offering a more robust alternative to Redis's data handling.

For associating names like variables or functions with their definitions, RedType uses static binding. A key factor in the decision to use static binding is that RedType's query language, in its current design, will not support advanced datatypes such as classes with inheritance or virtual methods. These are the type of features that typically necessitate more complex dynamic binding mechanisms. Since RedType focuses on operations with schema-defined data and simpler constructs, static binding is a straightforward choice. This means the interpreter determines what a name refers to during the interpretation phase, leading to predictable code and allows the system to immediately report errors.

The visibility of names in RedType is determined by static scope (also called lexical scope). This means a variable's scope, which is where it can be used, is set by its location in the code structure, like inside a function. Lexical scope makes code easier to read and maintain because developers can see where a variable comes from and where it applies by looking at the code. It also helps avoid unexpected behaviors and name conflicts, which can be issues in languages with different scoping rules (see Section 4.4).

Together, these basic design choices help make the RedType Query Language robust, more predictable, and familiar for developers used to statically-typed, lexically-scoped languages.

5 Requirements Specification

To guide development of RedType, we use a MoSCoW model to prioritize requirements. Since the focus of this initial MVP is the language and type system, "Must have" items cover core linguistic and key-value features. More advanced server capabilities and data structures are considered lower priority (Should/Could have) or deferred (Won't have).

The meaning of each MoSCoW requirement type has been defined as such for this project:

- **Must have**: Core functionality required for the language and basic server viability. Implementation depends on technical feasibility within project time constraints.
- **Should have**: Important enhancements that add significant value but aren't critical for the initial language-focused deployment.
- **Could have**: Desirable features with clear implementation paths that may be deferred.
- **Won't have**: Explicitly excluded features that go beyond the scope of the project phase.

This model helps us focus development efforts on critical language functionality while maintaining a clear roadmap for future enhancements. Each requirement is deliberately formulated to be objectively measurable to facilitate accurate assessment of fulfillment. The requirements table below is sorted by MoSCoW priority.

Table 1: MoSCoW Requirements for RedType Implementation

ID	MoSCoW	Requirement
1	Must have	The system must provide a schema definition language that supports primitive types (String, Int, Double/Float, Boolean).
2	Must have	The schema definitions must be enforceable on the server-side before query interpretation and execution.
3	Must have	The system must implement a type-safe query language that can detect type errors before query interpretation.
4	Must have	The server must provide a key-value data store that supports all defined primitive data types with consistent type enforcement.
5	Must have	The query language must support CRUD operations (Create, Read, Update, Delete) on all supported data types.

Continued on next page

Table 1 – *Continued from previous page*

ID	MoSCoW	Requirement
6	Must Have	The query language must support filtering operations with multiple conditions using comparison and logical operators.
7	Must Have	The query language must support field selection to limit the returned data.
8	Must have	Queries must be executed atomically to prevent race conditions and ensure data consistency.
9	Should have	The schema definition language should support collection types (Set, List, HashMap, Sorted List).
10	Should have	The system should implement periodic snapshots to persist data across server restarts.
11	Should have	The system should support concurrent atomic queries by implementing key-level locking instead of global locks.
12	Could have	The system could support different response encodings (e.g., base64) for binary data.
13	Could have	The system could support a Binary primitive type for storing raw byte sequences.
14	Could have	The system could implement multithreading to handle concurrent client connections and operations efficiently.
15	Could have	The system could have rollbacks in case atomic operations fail.
16	Could have	The system could support nested types to allow for hierarchical data structures.
17	Won't have	The system won't support reference types that allow relationships between defined schemas.
18	Won't have	The system won't support schema migrations in this phase.
19	Won't have	The system won't implement distributed database features like sharding or replication in this project phase.
20	Won't have	The system won't have the Pub/sub features Redis currently supports in this project phase.

Must Have Requirements (ID 1-8)

These requirements establish the core foundation of the RedType language and its basic execution environment. They encompass the essential primitive types, enforce schema integrity on the server side, define the type safe query language itself, and provide a fundamental key-value store for primitive values. They also enable basic data manipulation through CRUD operations, support key query optimization features such as filtering and field selection, and ensure atomic execution of queries. Together, these elements form the basis for a consistent and type safe language.

Should Have Requirements (ID 9-11)

These requirements represent important features for a more complete and robust Redis-like system but are secondary to the initial language definition focus. This includes support for common Redis collection types, which adds significant data modeling capabilities but extends beyond the basic key-value core initially prioritized. Data persistence via snapshots, and improved concurrency with key-level locking also fall into this category, enhancing functionality and reliability towards a production-ready solution.

Could Have Requirements (ID 12-16)

These are desirable features that offer additional flexibility or robustness but are not essential for the current project phase. They include support for binary data encoding options, the binary type itself, server performance optimizations like multithreading, enhanced transaction safety with rollbacks, and advanced data modeling with nested types. These can be considered for future iterations.

Won't Have Requirements (ID 17-20)

These features are explicitly excluded from this project phase. Reference types are omitted, as this shifts towards a relational model, diverging from the Redis-like philosophy. Schema migrations are deferred as the focus is on defining the language, not its long-term operational management. Complex distributed features like sharding or replication are also out of scope to ensure focus on the core single-instance type system. Although Pub/Sub is a prominent feature in Redis, it will be excluded from the system in this project phase due to the limited scope. However, it could be considered for future implementation once the core features are in place.

These requirements help define the key areas of the RedType language and provide a baseline for its development. This ensures the language evolves with clarity, consistency, and extensibility in mind.

6 Design

This section presents the design of the RedType language. It outlines key decisions made in the development of the language, including the definition of core commands, strategies for handling non-existent keys, and the use of constructs such as option types. Furthermore, it addresses important considerations such as deadlock prevention and the distinction between mutable and immutable variables.

The section proceeds with a detailed program example that demonstrates how RedType can be applied in practice. This example serves as a foundation for understanding the language's abstract syntax, which is introduced next. Finally, the concrete syntax is presented to bridge the conceptual design with its actual implementation.

Together, these components provide a comprehensive overview of RedType's design, offering insight into both its structure and its practical use.

6.1 RedType Schema

The design of schemas in RedType draws inspiration from schema definitions in other languages. As discussed in section 2.5, SQL uses a straightforward model where a table is defined by an identifier, followed by a list of fields. Each field has a specified type and may include constraints (see listing 5).

To maintain familiarity and simplicity for users, RedType adopts a similar structure. A schema in RedType begins with an identifier, followed by a block of field declarations. Each field is associated with a type to ensure type safety and prevent mismatches. Additionally, RedType uses the `@primary` annotation to indicate which field serves as the primary identifier for records of that type.

An example schema is shown below in listing 7, representing a `Product` with five fields:

```
1 Product {
2   id: String @primary,
3   name: String,
4   price: Double,
5   stockAvailable: Int,
6   stockReserved: Int
7 }
```

Listing 7: RedType Schema Example

This schema defines the structure of all `Product` records. The `id` field is marked with `@primary`, indicating it serves as the unique identifier. Each field is followed by its type, specifying the expected type of data to it. Although this example shows a single record

structure, in practice, a RedType schema can define many such record types, and each type can be instantiated with multiple data entries.

A crucial aspect of RedType is its approach to data addressing through structured keys. In Redis, keys are flat strings used to access values, for example, `"user:1000"`. While this is simple and flexible, it lacks type awareness and structure. RedType builds upon this model by introducing keys that are both precise and type-safe.

In RedType, a key refers to a specific field within a schema-defined record. This structured addressing allows precise access while preserving the guarantees provided by the schema. For example:

```
1 Product["itemId"].name
```

In this key expression:

- `Product` refers to the record name.
- `"itemId"` identifies a particular record by its primary key.
- `name` is the field being accessed within that record.

The equivalent Redis-style key might look like `"product:itemId:name"`, and while the string-based approach is simple, RedType formalizes this structure, making it part of the type system.

6.2 Core Command Set

The RedType Query Language includes command-like operations, inspired by those in Redis, as built-in parts of the language itself. This design means that unlike Redis, where Lua scripts execute distinct server commands (e.g., via `redis.call()`), RedType's command-like operations are an integral part of the Query Language itself. Consequently, all such operations are inherently subject to type checking against the schema during the interpretation or compilation of the RedType script. This integrated method ensures that all data manipulations are type-safe and conform to the defined schema. These operations are currently embedded within scripts, though future development might allow them to function as standalone commands.

The commands listed below illustrate this. Each description explains the command's function, its syntax, and how it operates within RedType's type safe environment using keys.

Foundational Data Commands

These are fundamental RedType commands. They are inspired by common operations in Redis, are used for interacting with specific type safe data fields in RedType. Data is accessed via specific structured identifiers as defined above. All operations strictly adhere to the schema. These commands can all be seen in the table below.

Command	Description
`SET <key> TO <value>`	Assigns a value to a field. Ensures the value conforms to the schema-defined type for the key.
`GET <key>`	Retrieves the value of the specified field. Returns `None` (as an `Option`) if the key is unset.
`DEL <key> [<key> ...]`	Deletes one or more field values. Subsequent `GET` returns `None`.
`INCR <key> [BY <increment>]`	Increments a numeric field (`Int` or `Double`) by the specified value (default is 1). Sets missing values to 0 before incrementing.
`DECR <key> [BY <decrement>]`	Decrements a numeric field by the specified value (default is 1). Type safety enforced as with `INCR`.

Table 2: Summary of RedType Commands

6.3 Handling Non-Existent Keys

A fundamental question for RedType is how to handle requests for data that is not there. When a script attempts to `GET` a key that has not been previously `SET` or has been deleted, the system needs a clear and consistent way to respond. This behavior directly impacts how developers write script logic, handle potential issues, and reason about the data within RedType's type safe environment.

This section explores several potential strategies for managing these situations during query execution. It outlines how each approach works and its implications.

Returning a Special `null` Value

One approach is to return a special `null` value when a `GET` operation targets a non existent key. This `null` is a distinct marker signifying absence.

The mechanism involves the `GET` operation returning this `null` marker. The script logic must then explicitly check if the returned value is this specific `null` before attempting to use it as its expected type.

```
val: Int = GET myMissingKey;

if (val == null) { // Assumes 'null' keyword and comparison
  print("Key was not found.");
  // Handle missing key...
} else {
  print("Value found:", val);
  // Use 'val' as an integer
}
```

Listing 8: Script checking for null

Regarding implications, this approach benefits from familiarity, as `null` is widespread. It is straightforward to implement. However, it can potentially weaken strict type safety if scripts do not diligently check for `null`. This risks runtime errors if `null` is used inappropriately, and places the safety burden on careful programmer checks within the script.

Raising a Runtime Error

An alternative is to treat requesting a non existent key as a runtime error within the execution environment. The `GET` operation fails and triggers an error handling path or halts the script.

The mechanism involves generating a specific error (for example `KeyNotFoundError`) if the key is missing during a `GET` call. If the script does not handle this error perhaps using `try...catch`, the error would typically abort the script. It would then be propagated back to the client application.

```
result: Int = -1; // Default value in case of error

try { // Assumes try-catch mechanism exists in PL
  val: Int = GET myMissingKey;
  print("Value found:", val);
  result = val; // Assign if successful
} catch (KeyNotFoundError) {
  print("Key not found. Using default.");
  // Error handled within the script
}

// Continue script execution...
```

Listing 9: Script handling a potential error

For implications, treating a missing key as an error makes it clear that something went wrong, and the script must handle this situation, Encouraging the writing of more reliable code. This approach also maintains type purity. A variable declared as `int` will only hold an `int` if `GET` succeeds. The main drawback is potential verbosity. Requiring error handling statements might clutter script logic, especially if missing keys are frequent and expected.

Using Option Types (`Option<T>`)

This approach uses the type system itself to represent potential absence. The `GET` operation returns an `Option<T>`, not the value directly (`T`). This wrapper type explicitly indicates whether a value is present (`Some(T)`) or absent (`None`).

The mechanism relies on the type signature of `GET` making absence explicit. The Program Language requires constructs like `match` or specific methods, to safely access the value inside the `Option<T>`. This forces the script logic to handle both possibilities.

```
maybe_val: Option<Int> = GET myMissingKey; // GET returns Option<int>

match maybe_val { // Assumes Option<T> and match syntax in PL
  Some(val) => { // 'val' is guaranteed to be int here
    print("Value found:", val);
    // Use 'val' within the script
  }
  None => {
    print("Key was not found.");
    // Handle absence within the script
  }
}
```

Listing 10: Script using Option types

Considering the implications, `Option` types offer the highest level of type safety regarding missing values. By encoding absence into the type system, they force the script to handle both cases during interpretation. This effectively eliminates runtime errors caused by unexpected missing data. While robust, this pattern might be less familiar. It can also introduce verbosity through the required matching or unwrapping logic in the script. It necessitates adding `Option` types and handling mechanisms to the Program Language.

Providing a Default Value via Schema

This approach defines a default value directly within the data schema. When a `GET` operation targets a key associated with a field that has a defined default, and the key itself does not exist, the operation automatically returns the predefined default value from the schema.

The mechanism involves specifying the default in the schema. The `GET` operation consults the schema. If the key is absent and a default exists, that default value is seamlessly provided to the script logic.

```
// Part of the RedType Schema Definition
type UserProfile {
  userId: String @primary,
  visitCount: Int @default(0), // Default value specified here
}
```

Listing 11: Schema Definition with a default value

```
// Assume 'user[123].visits' key does not exist, but schema defines visitCount:
    Int @default(0)

visits: Int = GET user[123].visits;

// 'visits' will automatically be 0 due to the schema default.
print("User visit count (or default): ", visits);
```

Listing 12: Script using GET with schema default

The implications are that defaults become part of the formal data contract, ensuring consistency. Script logic becomes simpler as it does not need explicit default handling for `GET`. However, this removes flexibility from the query logic. The default is fixed in the schema. Crucially, this approach masks the fact that the key was actually missing. This might obscure important information needed by the script logic. It prevents distinguishing a genuinely stored default value from a missing key situation that resulted in the default.

Conclusion: Selecting Option Types for Purity and Safety

Based on the analysis in section 6.3, RedType will implement Option types for handling non-existent keys. This choice is driven by the pursuit of the purest, and most robust type safety model.

By encoding the potential absence of a value directly within the type system (representing results as either `Some(T)` or `None`), Option Types mandate explicit handling of both possibilities at the language level during interpretation. This eliminates the risk of runtime

errors that can occur in other approaches, such as developers forgetting to perform null checks. It ensures that the type system itself prevents the misuse of potentially absent data, aligning with RedType's core objective of maximizing safety and predictability.

6.4 The Option Type Philosophy

Having chosen Option Types as the mechanism for handling potentially absent values, it is beneficial to explore the underlying philosophy driving this decision, particularly drawing parallels with languages like Rust where this pattern is central. This approach strongly aligns with RedType's goal of enhancing type safety compared to traditional key-value stores.

The motivation originates from a fundamental critique of `null` references, prevalent in many languages. Tony Hoare, the inventor of the `null` reference, famously called it his "billion-dollar mistake" [48]. He stated, "My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a `null` reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years" [48]. [16] The core problem lies in the implicit nature of `null`: in many languages, any reference variable can potentially hold `null`, forcing developers into constant defensive checks and risking runtime errors (like `NullPointerException` or segmentation faults) when `null` is used incorrectly. [16] [5]

Rust, aiming for robust safety guarantees, explicitly excludes the `null` feature. Instead, it adopts the `Option<T>` Enum, defined in its standard library, to encode the possibility of absence directly into the type system. This Enum has two variants: `Some(T)`, which indicates that a value of type `T` is present and wraps that value; and `None`, which indicates that no value is present [16] [5].

The crucial difference lies in type system enforcement. In Rust, `Option<T>` is an entirely distinct type from `T`. The compiler prevents using an `Option<T>` where a `T` is expected. This forces the programmer to explicitly handle the `None` case before accessing the potential inner value, typically using constructs like `match` or methods like `unwrap_or`. This static check, conducted before runtime, effectively eliminates the entire class of errors associated with unexpected null values.

By adopting this philosophy for handling non-existent keys, RedType aims to provide similar safety benefits. It mandates explicit handling of potential absence at the language level, preventing runtime errors and making the code more predictable and reliable, which is central to RedType's objective of maximizing type safety.

6.5 Mutability in RedType

In modern programming languages, the handling of variables plays a crucial role in determining the flexibility and efficiency of the language. The decision to use mutable or immutable variables in certain scenarios can affect performance, usability, and the overall behavior of the language.

RedType adopts mutable variables as the default to support efficient and flexible data manipulation during program execution. This design decision is particularly important in the context of dynamic program logic, where variable states often change in response to user actions or query results.

Allowing variables to be updated in place reduces the overhead of repeatedly creating new instances, which would be required in an immutable model. This choice is aligned with RedType's goal of supporting real-time, stateful operations while keeping execution performance high.

Argument Passing Mechanisms

One crucial aspect of designing a language is choosing between using pass-by-reference, pass-by-value, and pass-by-name. In RedType, the chosen mechanism is pass-by-value for all types except arrays. This design choice supports several key goals. Primarily, pass-by-value promotes data immutability, ensuring that functions cannot unintentionally modify the original data. This aligns with RedType's emphasis on clarity and predictable behavior, making it easier to understand and reason about how functions operate. Moreover, it avoids the complexities of pass-by-reference, such as unintended side effects or conflicts when multiple functions modify the same variable. By isolating variables within the scope of each function, pass-by-value contributes to a safer and more reliable execution model.

For arrays pass-by-reference has been chosen to enable efficient manipulation through built-in functions such as push and pop. This design decision also addresses performance considerations in scenarios involving large arrays. For instance, when working with an array containing 10,000 elements, pass-by-value would require copying the entire array each time it is used in a query. Any modification to a single element would necessitate creating and overwriting the entire array, resulting in significant computational overhead. In contrast, pass-by-reference allows operations to directly modify the original array, eliminating the need for redundant data copying. This approach significantly reduces memory usage and improves execution efficiency, especially in data-intensive applications

The choice of combining pass-by-value and pass-by-reference in RedType ensures that the language maintains its emphasis on clarity, safety, and predictable behavior, while also addressing the practical need for efficiency in handling large data structures.

6.6 Program Example

This section builds directly on the Redis-based inventory example presented in section 2.2, where Redis was used to reserve and confirm product stock during a high-traffic sales event such as Black Friday. Such events place enormous stress on systems. They require rapid processing of concurrent requests while ensuring data consistency, reducing mistakes like overselling. RedType's design emphasizes server side execution of type safe logic with explicit locking. This helps address these challenges effectively.

The example in this section, demonstrates defining a clear data structure using RedType's Schema Definition Language. It shows implementing specific key locks for atomic operations and it details crafting server side scripts with functions for efficient execution. It also shows the synergy of these components in providing a reliable inventory management solution.

It is important to note that while this example addresses a realistic scenario the provided RedType code is illustrative and conceptual. A production system for a high traffic store would involve further considerations and complexities. For instance features like managing expiring shopping cart reservations handling payment gateway integrations and more sophisticated error recovery mechanisms are essential in a real world application but have been excluded here to maintain focus on RedType's core language features and server side scripting capabilities.

Defining the Data Structure: The Store Schema

Before any operations can occur we must define the structure of our data. For our inventory system we need to track essential details about each product. This includes its available stock and any stock that has been reserved by customers.

```
Product {
  id: String @primary,
  name: String,
  price: Double,
  stockAvailable: Int,
  stockReserved: Int
}
```

Listing 13: Store Schema

In this `Product` schema `id:String @primary` is a unique identifier for the product serving as its primary key. The `name: String` stores the product's name and `price:Double` holds its price. `stockAvailable:Int` represents the total physical quantity of the product in stock. Finally `stockReserved:Int` indicates how many units of this product are currently in users shopping carts or otherwise temporarily held. The actual number of items

available for a new reservation is calculated from `stockAvailable - stockReserved`. This distinction is crucial for accurate inventory display and preventing overselling.

Implementing Core Operations with RedType Scripts

With the schema in place we can define the operational logic. In RedType this logic is encapsulated in scripts that are sent to the RedType server for execution.

A RedType script defines an operation to be performed on the server. Such a script can include multiple `LOCK` declarations for data consistency. If a script attempts to access an unlocked key, then a rollback will occur.

It might optionally include one or more function definitions to organize the logic. Then it has the main sequence of statements that perform the work. This main sequence may conclude with a `return` statement if a specific value needs to be sent back to the client though returning a value is optional. If a script does not explicitly return a value the client would still receive a general acknowledgment of the script's execution status from the server.

The following sections showcase examples of RedType use cases, as introduced in 2.2, where a simplified product reservation and purchase confirmation were demonstrated. Query Example 1, presented below illustrates how the logic is expressed in RedType, while additional examples are provided in the appendix A.1, A.2 and A.3

Query Example 1: Attempting to Reserve a Product

This script is executed when a customer attempts to add a hypothetical Black Friday special item, with the ID `"bf_special_item_001"` to their shopping cart. It checks for availability and if possible reserves the requested quantity. The `GET` operations return an `Option` type which is then handled using a `match` statement. This is RedType's way of ensuring that the potential absence of data is explicitly managed preventing errors.

```
1  LOCK Product["bf_special_item_001"].stockAvailable,
2       Product["bf_special_item_001"].stockReserved;
3
4  func attemptToReserveProduct(productId: String,
5  quantityToReserve: Int): String {
6
7      stockAvailableOpt: Option<Int> = GET Product[productId].stockAvailable;
8      currentStockAvailable: Int = 0;
9
10     match stockAvailableOpt {
11         Some(sa) => {
12             currentStockAvailable = sa;
13         }
```

```
        None => {
            return "FAILURE: Product ID not found.";
        }
    }

    stockReservedOpt: Option<Int> = GET Product[productId].stockReserved;
    currentStockReserved: Int = 0;

    match stockReservedOpt {
        Some(sr) => {
            currentStockReserved = sr;
        }
        None => {
            return "FAILURE: Product reservation data error.";
        }
    }

    effectiveStockForReservation: Int =
    currentStockAvailable - currentStockReserved;




    if (quantityToReserve > 0 &&
    quantityToReserve <= effectiveStockForReservation) {
        INCR Product[productId].stockReserved BY quantityToReserve;
        return "SUCCESS: Items reserved.";
    }
    else {
        if (quantityToReserve <= 0) {
            return "FAILURE: Quantity to reserve must be positive.";
        }
        else {
            return "FAILURE: Insufficient stock to reserve requested quantity.";
        }
    }
}

reservationStatus: String =
attemptToReserveProduct("bf_special_item_001", 1);

return reservationStatus;
```

Listing 14: RedType Script: Reserve Product

When this script is sent to the server it first acquires locks on the specified fields for product `"bf_special_item_001"`. It then defines and calls the `attemptToReserveProduct` function. The function's return value indicating success or failure is then returned by the script itself to the calling client. This ensures the reservation logic is atomic for this item.

Query Examples Explained

The following examples summarize query scripts that build on top of the previous examples. These are included in the appendix:

- **Query Example 2 - Confirming a Purchase** After a successful reservation and payment, this script confirms the purchase by decrementing both the reserved and available stock fields. It returns a confirmation message. See appendix A.1
- **Query Example 3 - Administrative Stock Update** This script adjusts the available stock directly, such as when restocking. It returns a status message. See appendix A.2
- **Query Example 4 - Generating a Stock Level Report** This function calculates a stock report and returns a formatted string showing the computed stock level using the built-in `numericToString` function for conversion. See appendix A.3

Integrating Queries into an Application Workflow

These individual RedType query scripts form the core server side components that an application uses to interact with the RedType database. An application backend such as a web server developed in a language like TypeScript, Python or Java would send these scripts to the RedType server, and process the responses. This communication would be managed by a RedType client library designed for a specific programming language.

To illustrate, here is a conceptual TypeScript example. It shows how an application might send the content of one of the previously described RedType scripts to the RedType server.

```
import { RedTypeClient } from "redtype-client"; // Hypothetical library

const client = new RedTypeClient("http://redtype.server:port");

async function executeServerScript(scriptContent: string): Promise<string> {
  console.log("Sending script to RedType server...");
  try {
    const resultFromServer = await client.executeScript(scriptContent);
    console.log("RedType server responded:", resultFromServer);
    return resultFromServer as string;
  } catch (error) {
    console.error("Error executing RedType script:", error);
    return "CLIENT_ERROR: Script execution failed.";
  }
}

const scriptForReservation = `Query Example 1 Inserted Here...`;

async function handleUserAction() {
  const reservationStatus = await executeServerScript(scriptForReservation);
  if (reservationStatus === "SUCCESS: Items reserved.") {
    console.log("Reservation successful application can proceed.");}
  else {
    console.log("Reservation failed. Status:", reservationStatus);}
}

handleUserAction();
```

Listing 15: Conceptual Client Side Interaction (TypeScript Example)

This client side code snippet illustrates sending a self contained RedType script to the server. The `scriptForReservation` variable would hold the entire script e.g Query Example 1 from section 6.6. The server executes this script atomically with respect to the locked keys and then returns the result. In a real application this script content might be loaded from a file or template rather than being an inline multiline string for better maintainability.

By adopting this model of a well defined schema targeted server side scripts with specific locks and clear client interaction patterns RedType offers a way to build reliable applications. These applications are capable of handling demanding scenarios like inventory management during peak sales periods.

6.7 RedType Language Syntax

RedType will consist of two integrated sub-languages: the Schema Definition Language, used to define data structures, and the Program Language, used to express executable logic such as functions and database operations. In the following section, we present the Abstract and Concrete Syntax of RedType, showing how the two sub-languages are combined into a unified language.

Before defining the abstract syntax for RedType, it is worth explaining what abstract syntax actually is. Abstract syntax describes the structure of a program without concerning itself with how the code is specifically written. Operator precedence, parentheses, etc. are all ignored, and instead, we focus only on the essential components that define how a program is built.

In Figure 2 and Listing 16 below the abstract and concrete syntax for RedType can be seen. This syntax includes all formation rules that define how the data types, queries, and constraints will be structured within the language. The abstract syntax uses the notation $\overrightarrow{x}$ when an element (x) is included an undefined amount of times and the concrete syntax uses the following notations:

- ´?´ is used when an element can be included "zero or one time" in a formation.
- ´*´is used when an element can be included "zero or more times" in a formation.
- ´+´ is used when an element can be included "one or more times" in a formation.

Abstract Syntax

An abstract syntax for a program is defined by: First dividing the language into different collections of syntactic categories and then for each of these categories, specifying a finite set of formation rules that describe how elements within that category can be constructed [12, p.27]. Figure 2 presents the combined abstract syntax of the RedType Language, as outlined in Section 6.11, followed by an abstract syntax tree derived from an expression.

$$
\begin{aligned}
P \in \mathit{Program} &::= \overrightarrow{S}\ \overrightarrow{L}\ \overrightarrow{S_t} \\
S \in \mathit{SchemaDefinition} &::= \overrightarrow{T_d} \\
T_d \in \mathit{TypeDefinition} &::= x\ \overrightarrow{F} \\
F \in \mathit{FieldDefinition} &::= x : T\ \overrightarrow{Con} \\
\mathit{Con} \in \mathit{Constraint} &::= \texttt{@primary} \\
L \in \mathit{Lock} &::= \texttt{LOCK}\ \overrightarrow{K} \\
S_t \in \mathit{Stmt} &::= x : T = e \mid x = e \mid \texttt{if}\ b\ \texttt{then}\ S_{t1}\ \texttt{else}\ S_{t2} \\
&\quad \mid \texttt{if}\ b\ \texttt{then}\ S_t \mid \texttt{for}\ x\ \texttt{in}\ e\ S_t \mid \texttt{while}\ b\ \texttt{do}\ S_t \mid D_{op}; \\
&\quad \mid \mathit{skip}; \mid \texttt{return}\ e \mid \texttt{return} \mid e; \mid \texttt{match}\ e\ \overrightarrow{Case} \\
&\quad \mid \texttt{func}\ x(\overrightarrow{x : T}) : T\ S_t \mid \texttt{func}\ x(\overrightarrow{x : T})\ S_t \\
\mathit{Case} \in \texttt{MatchCase} &::= \texttt{Some}(x)\ S_t \mid \texttt{None}\ S_t \\
K \in \mathit{Key} &::= T_d\ [s]\ F \\
D_{op} \in \mathit{DatabaseOp} &::= \texttt{DEL}\ \overrightarrow{K} \mid \texttt{INCR}\ \overrightarrow{K} \mid \texttt{INCR}\ \overrightarrow{K}\ \texttt{BY}\ e \\
&\quad \mid \texttt{DECR}\ \overrightarrow{K} \mid \texttt{DECR}\ \overrightarrow{K}\ \texttt{BY}\ e \mid \texttt{SET}\ K\ \texttt{TO}\ e \\
&\quad \mid x : \texttt{Option<}C_t\texttt{>} = \texttt{GET}\ K \mid x = \texttt{GET}\ K \\
T \in \mathit{Type} &::= C_t \mid \texttt{Option<}C_t\texttt{>} \\
C_t \in \mathit{CoreType} &::= P_t \mid P_t[\,] \\
P_t \in \mathit{PrimitiveType} &::= \texttt{Int} \mid \texttt{Double} \mid \texttt{String} \mid \texttt{Bool} \\
e \in \mathit{Expr} &::= (e) \mid \texttt{true} \mid \texttt{false} \mid n \mid d \mid s \mid x \mid x(\overrightarrow{e}) \mid [\overrightarrow{e}] \\
&\quad \mid \texttt{Some}(e) \mid \texttt{None} \mid e_1\texttt{BinOp}\ e_2 \mid e_1\texttt{RelOp}\ e_2 \mid \texttt{UnaryOp}\ e_1 \\
\mathit{BinOp} \in \mathit{BinOp} &::= + \mid - \mid * \mid / \mid \% \mid \hat{} \mid \wedge \mid \vee \\
\mathit{RelOp} \in \mathit{RelOp} &::= < \mid > \mid == \mid \neq \mid \leq \mid \geq \\
\mathit{UnaryOp} \in \mathit{UnaryOp} &::= \neg \mid + \mid - \\
\\
x \in \mathit{Identifier}, &\quad n \in \mathit{Integer}, \quad d \in \mathit{Double}, \quad s \in \mathit{String}
\end{aligned}
$$

Figure 2: Abstract Syntax for the RedType Language

The abstract syntax in Figure 2 defines the structural elements of the RedType language, encompassing both schema definitions and the programming language, independent of any concrete notation. It specifies the core constructs for building programs, schemas, functions, statements, expressions, and types.

Overall Program Structure (P)
At the highest level, a RedType program (P) follows the sequence $P ::= \overrightarrow{S}\ \overrightarrow{L}\ \overrightarrow{S_t}$. This specific structure ensures that Schema Definitions ($\overrightarrow{S}$) are processed first, followed by any Lock Declarations ($\overrightarrow{L}$) needed for concurrent access control, and finally the executable Statements ($\overrightarrow{S_t}$) containing the program logic. This ordering establishes the data context and concurrency controls before execution begins.

Schema Definition (S, T_d, F, Con)
Schema Definitions (S) are composed of Type Definitions ($\overrightarrow{T_d}$), which in turn define named structures using Fields ($\overrightarrow{F}$). Each Field Definition (F) associates a name with a specific data type (T) and may include optional constraints ($\overrightarrow{Con}$). The defined syntax explicitly includes `primary`) as an example constraint, used to designate primary identifier fields.

Lock Declarations (L) and Atomicity
A key aspect for ensuring safe concurrent operations is the `Lock` declaration ($L ::= \texttt{LOCK}\ \overrightarrow{K}$). As detailed in the deadlock prevention strategy (Section 6.9), these declarations are fundamental to RedType's atomicity guarantees. The expression $\overrightarrow{K}$ specifies keys that the subsequent code intends to access. Crucially, the RedType runtime must acquire exclusive locks on the union of all keys listed across all $\overrightarrow{L}$ declarations before executing any code Statements (S_t). This pre-emptive locking strategy prevents deadlocks arising from dynamic lock acquisition and ensures that `Statements` (S_t) are executed atomically. Any attempt by the code to access a key not pre-declared in L results in a runtime failure and rollback.

Statements (S_t)
The actual program logic resides within executable Statements (S_t). Statements dictate the actions performed during execution. RedType includes standard statement forms such as variable handling (declaration with $x : T = e$, assignment $x = e$), conventional control flow constructs (`if`, `for`, `while`), a no-operation statement (`skip`), and return statements (`return`). More specific to RedType's design are the mechanisms for type safety and database interaction. The `match` statement is essential for safely handling the `Option<T>` type (discussed in sections 6.3 and 6.4), requiring explicit code paths for both the `Some(x)` case (accessing the value) and the `None` case (handling absence). Furthermore, Database Operations (D_{op}) are integrated directly as statement types including `DEL`, `INCR/DECR`, `SET` and `GET`. Notably, the `GET` operation returns an `Option<T>` value, enforcing safe access to potentially non-existent keys and supporting core CRUD functionality. Lastly, Statements

(S_t) also cover Function Definitions (D). Functions provide modularity and are defined using the `func` keyword, followed by a name, typed parameters, an optional return type, and a statement (S_t) serving as the function body.

Type System (T)
RedType's type system is central to its goal of enhanced safety. The `Type` category (T) defines the valid data types, enforced during interpretation and execution. The Type (T) consists of the Core Types (C_t) and the crucial Option Type ($Option < C_t >$) discussed in (section 6.4). The Core Type (C_t) then consist of either primitive types (P_t) or arrays made of `Primitive Types` ($P_t[]$) indicated by a primitive type followed by square brackets. The foundational `Primitive Types` (P_t) include `Int`, `Double`, `String`, and `Bool`. The Options Type ($Option < C_t >$) constructor wraps any other Core Type C_t to explicitly represent the potential absence of a value, distinguishing between `Some(value)` and `None`, thereby forcing safe handling of optional data through mechanisms like the `match` statement.

Expressions ($Expr$)
Expressions denoted by e represent computations that produce values. RedType employs a unified expression category supporting various data types. Expressions include standard elements like literals (numeric n, d, string s, boolean `true`, `false`), variables (x), function calls ($x(\overrightarrow{e})$) and array literals ($[\overrightarrow{e}]$). Specific to handling optionality, expressions include constructors for option types: ($Some(e)$) to wrap a value and $None$ to signify absence. Expressions include arithmetic expressions (`BinOp`), comparisons (`RelOp`) and `Unary` operations (`UnaryOp`). A standard range of operators is supported, categorized as `BinOp` $(+, -, \dots)$ for operations on two values, `RelOp` $(<, ==, \dots)$ for comparing two values, and `UnaryOp`$(\neg, \wedge, \dots)$ for declaring signs and negating values. The relational (`RelOp`) and Unary (`UnaryOp`) operators are particularly important for building conditions used in control flow statements, enabling data filtering capabilities as required by Requirement 6. Lastly, parentheses (e) can be used to control the evaluation order as expected.

Concrete Syntax

While the abstract syntax defines the structural rules and formal grammar of the RedType Language, the concrete syntax specifies how these structures are written in practice. It provides the exact tokens, keywords, and formatting conventions that developers use to define schema definitions, statements and so on in a EBNF-like format.

```
Program::= SchemaDefinition? Lock? Stmt*

// ===== SCHEMA =====
SchemaDefinition ::= TypeDefinition (',' TypeDefinition)*

TypeDefinition   ::= Identifier '{' FieldDefinition
                     (',' FieldDefinition)* '}'

FieldDefinition  ::= Identifier ':' PrimitiveType Constraint?

Constraint       ::= '@primary'

// ===== LOCK =====
Lock ::= LOCK Keyidentifer (',' KeyIdentifier)*

// ===== STATEMENTS =====
Stmt             ::= Identifier ':' Type '=' Expr ';'
                   | Identifier '=' Expr ';'
                   | Identifier '[' Expr ']' '=' Expr ';'
                   | 'if' '(' OrExpression ')' '{' Stmt* '}'
                     ('elif' '(' OrExpression ')' '{' Stmt* '}')*
                     ('else' '{' Stmt* '}')?
                   | 'for' Identifier 'in' Expr '{' Stmt* '}'
                   | 'while' '(' OrExpression ')' 'do' '{' Stmt* '}'
                   | DatabaseOp ';'
                   | 'skip' ';'
                   | 'return' Expr? ';'
                   | Expr ';'
                   | 'match' Expr '{' Case* '}'

FunctionDef      ::= 'func' Identifier '(' ParamList? ')'
                     (':' ReturnType)? '{' Stmt* '}'

ParamList        ::= Param (',' Param)*
```

```
Param            ::= Identifier ':' Type
ReturnType       ::= Type

Case             ::= 'Some' '(' Identifier ')' '=>' '{' Stmt* '}'
                   | 'None' '=>' '{' Stmt* '}'

DatabaseOp       ::= 'DEL' KeyIdentifier (',' KeyIdentifier)*a
                   | 'INCR' KeyIdentifier (',' KeyIdentifier)*
                   ('BY' Aexp)?
                   | 'DECR' KeyIdentifier (',' KeyIdentifier)*
                   ('BY' Aexp)?
                   | 'SET' KeyIdentifier 'TO' Expr
                   | Identifier '=' 'GET' KeyIdentifier
                   | Identifier ':' Type = 'GET' KeyIdentifier

KeyIdentifier    ::= Identifier '[' Value ']' '.' Identifier

Type             ::= CoreType | 'Option<' CoreType '>'

CoreType         ::= PrimitiveType
                   | PrimitiveType '[]'

PrimitiveType    ::= 'Int' | 'Double' | 'String' | 'Bool'

// ===== EXPRESSION =====
Expr ::= OrExpression

OrExpression ::= OrExpression '||' AndExpression | AndExpression

AndExpression ::= AndExpression '&&' CmpExpression | CmpExpression

CmpExpression ::= Aexp RelOp Aexp | Aexp

RelOp ::= '==' | '>' | '<' | '!=' | '>=' | '<='

Aexp ::= Aexp ('+' | '-') Term | Term

Term ::= Term ('*' | '/' | '%') Factor | Factor

```

```
74 Factor ::= UnaryOp Power | Power
75
76 UnaryOp ::= '!' | '+' | '-'
77
78 Power ::= AtomExpr '^' Power | AtomExpr // right-associative
79
80 AtomExpr ::= Identifier '(' ArgList? ')' | '[' ExprList? ']'
81           | 'Some(' Expr ')' | None | Atom
82
83 Atom ::= StringLiteral | IntLiteral | DoubleLiteral | Identifier
84          | BoolLiteral | '(' Expr ')'
85
86 // ===== GENERAL =====
87 ArgList          ::= Expr (',' Expr)*
88 ExprList         ::= Expr (',' Expr)*
89
90 Identifier       ::= [a-zA-Z_][a-zA-Z0-9_]*
91
92 Value            ::= DoubleLiteral | IntLiteral | StringLiteral
93                      | BoolLiteral
94
95 DoubleLiteral    ::= [0-9]+'.' [0-9]+
96 IntLiteral       ::= [0-9]+
97 StringLiteral    ::= '"' [^"]* '"'
98 BoolLiteral      ::= 'true' | 'false'
```

Listing 16: Concrete Syntax for RedType Language

Following the concrete syntax rules in Listing 16, two concrete schema definitions: User and Product, are illustrated below in Listing 17. Each include various field types, and constraint, demonstrating how schema authors can specify data models in a concise and structured way, using the schema definition sub-language.

```
type User {
  userId: String @primary,
  name: String,
  age: Int,
  isActive: Bool
},
type Product {
  productId: String @primary,
  price: Double,
  Stock: Int
}
```

Listing 17: Example of Concrete Schema Definition using the Concrete Syntax

Following the concrete syntax rules in Listing 16, a function consisting of a series of statements is seen below in Listing 16. The example demonstrates variable declarations, arithmetic expressions, conditional branching, and the use of return statements, showcasing how logic can be structured using the RedType program language.

```
func calculateBonus(salary: Double, performanceScore: Int): Double {
  bonusRate: Double = 0.10;
  bonus: Double = salary * bonusRate;

  if (performanceScore > 90) {
    bonus = bonus + 1000;
  } elif (performanceScore > 75) {
    bonus = bonus + 500;
  } else {
    bonus = bonus + 100;
  }

  return bonus;
}
```

Listing 18: Example of Function Definition using the concrete syntax in Listing 16

Together, the schema definition- and program sub language examples demonstrate how RedType's concrete syntax enables both the definition of structured data models and the expression of logic and computation. By unifying schema declarations with programmable constructs, RedType provides a language for specifying, manipulating, and interacting with typed data. This integration ensures that developers can work fluently across data modeling and programmatic logic within a single, consistent syntax.

6.8 Built-in functions

To enhance the capabilities of the RedType Program Language and provide common utilities, RedType includes a set of built-in functions. These functions offer predefined operations for common tasks, such as manipulating data collections and converting data types.

Array Handling

Arrays are a basic way in RedType to store a list of items where all items are of the same type. These arrays are mainly used to work with data in memory while a RedType script is running. They are useful for temporarily holding data, like gathering results from different steps, handling series of items, or storing data that is needed for a short time inside your script.

Declaring Arrays

In RedType, you declare an array by stating the type of items it will hold, followed by square brackets `[]`. You must set up an array when you declare it, either with some starting items or as an empty list. Importantly, all items in an array must be of the same declared type (e.g., all integers or all strings).

Syntax rules:

```
variableName :  PrimitiveType[] = [item1, item2, ...];
variableName :  PrimitiveType[] = []; // For an empty array
```

Examples:

```
1 // Declare and set up an array of whole numbers (integers)
2 active_user_ids : Int[] = [101, 203, 405];
3
4 // Declare an empty array for text (strings)
5 pending_tasks : String[] = [];
```

Listing 19: Syntax for array declaration and initialization

Modifying and Querying Arrays

RedType provides a set of built-in functions for working with arrays. These functions support common operations such as adding, removing, and accessing elements, as well as retrieving metadata like the array's length. The functions include `len`, `push`, `pop`, `get`, `insert`, and `removeAt`. Further details can be found in Appendix B. It is important to remember that arrays in RedType are pass-by-reference. This means when you pass an array to a function like `push` or `insert`, the function changes the original array that is in the script's memory.

String Conversion

RedType also provides built-in functions for converting values between numeric types and strings. These functions are particularly useful for input/output operations, data formatting, or when interacting with external systems that might provide numeric data as text.

The first function, `numericToString(value: Int | Double): String`, converts a numeric value (either `Int` or `Double`) into its string representation. This function always succeeds when given valid numeric input, and returns the string representation of the numeric value. For example, converting an integer to a string can be done by declaring a count variable as `count: Int = 123` and then using `count_str: String = numericToString(count)` to obtain the string `"123"`. Similarly, converting a double works by declaring `price: Double = 45.99` and using `price_str: String = numericToString(price)` to get `"45.99"`.

```
1 // Convert an integer to a string
2 count: Int = 123;
3 count_str: String = numericToString(count);
4 // count_str becomes "123"
5
6 // Convert a double to a string
7 price: Double = 45.99;
8 price_str: String = numericToString(price);
9 // price_str becomes "45.99"
```

Listing 20: Converting numbers to strings

The second function, `stringToInt(value: String): Option<Int>`, attempts to convert a given string value into an integer. This function only parses strings that represent valid integers. If parsing succeeds, it returns `Some(integer_value)`, and if parsing fails, it returns `None`. For instance, converting a valid integer string like `user_input_age: String = "30"` with `age_val: Option<Int> = stringToInt(user_input_age)` results in `Some(30)`. However, decimal strings are rejected for integer conversion, so `decimal_input:`

`String = "98.5"` converted with `decimal_val: Option<Int> = stringToInt(decimal_input)` becomes `None`. Similarly, invalid numeric strings like `user_input_invalid: String = "not_a_number"` also return `None` when processed.

```
// Convert a string to an integer
user_input_age: String = "30";
age_val: Option<Int> = stringToInt(user_input_age);
// age_val becomes Some(30)

// Decimal strings are rejected for integer conversion
decimal_input: String = "98.5";
decimal_val: Option<Int> = stringToInt(decimal_input);
// decimal_val becomes None

// Invalid numeric strings return None
user_input_invalid: String = "not_a_number";
invalid_val: Option<Int> = stringToInt(user_input_invalid);
// invalid_val becomes None
```

Listing 21: Converting strings to integers

The third function, `stringToDouble(value: String): Option<Double>`, attempts to convert a given string value into a double-precision floating point number. Unlike the integer conversion function, this function can parse both integer and decimal string representations. If parsing succeeds, it returns `Some(double_value)`, and if parsing fails, it returns `None`. Converting a decimal string such as `user_input_score: String = "98.5"` with `score_val: Option<Double> = stringToDouble(user_input_score)` results in `Some(98.5)`. Integer strings also work for double conversion, so `integer_input: String = "42"` converted with `int_as_double: Option<Double> = stringToDouble(integer_input)` becomes `Some(42.0)`. However, invalid strings still return `None`, as demonstrated with `invalid_input: String = "not_a_number"`.

```
// Convert a decimal string to a double
user_input_score: String = "98.5";
score_val: Option<Double> = stringToDouble(user_input_score);
// score_val becomes Some(98.5)

// Integer strings also work for double conversion
integer_input: String = "42";
int_as_double: Option<Double> = stringToDouble(integer_input);
// int_as_double becomes Some(42.0)
```

```
10
11 // Invalid strings return None
12 invalid_input: String = "not_a_number";
13 invalid_val: Option<Double> = stringToDouble(invalid_input);
14 // invalid_val becomes None
```

Listing 22: Converting strings to doubles

These string conversion functions enhance RedType's ability to process and manipulate data that may not initially be in the desired format, providing essential flexibility for script developers.

6.9 Preventing Deadlocks

Deadlocks are a serious problem in concurrent systems. They happen when multiple processes or threads compete for exclusive access to shared resources and get stuck. To ensure RedType queries run reliably and atomically, we need a solid plan for handling concurrency and preventing these deadlocks. This section will first look at optimistic and pessimistic concurrency control. Then, it will explain RedType's method for preventing deadlocks.

Optimistic and Pessimistic Control Approaches

Concurrency control strategies help keep data correct and the system stable when many queries use shared data at the same time.

Optimistic Concurrency Control (OCC) assumes that conflicts between queries are rare. In OCC, queries might operate on snapshots or private copies of data without initially acquiring locks. Before a query's modifications are finalized, the system verifies if the underlying data has been changed by other concurrent queries since it was read. If a conflict is detected, for instance, if version numbers show the data has changed, the query is usually rolled back and might need to be retried [2]. OCC can lead to faster performance when there are few conflicts because it avoids the overhead of locks. However, if conflicts are frequent, the system can slow down due to many rollbacks and wasted effort.

Pessimistic Concurrency Control (PCC), on the other hand, assumes conflicts are common. It makes queries get locks on data before they can work with it. If another query already has a conflicting lock, the new query must wait [2]. This "better safe than sorry" method aims to prevent conflicts before they occur. With PCC, once locks are acquired, the query's logic for data access is often simpler because it does not need to handle simultaneous modification conflicts for those locked resources. The system manages any necessary waiting for locks to become available. A general characteristic of PCC is that

it can reduce the number of queries that run concurrently compared to more optimistic strategies. Additionally, some PCC implementations can lead to deadlocks if lock acquisition is not managed carefully. However, RedType employs a specific PCC strategy using pre-declaration of all locks, which will be covered in section 6.9.

For RedType, the choice of concurrency control is primarily guided by its expected usage patterns. RedType anticipates scenarios where queries will frequently attempt to access and contend for the same keys. A prime example is the Black Friday sale, where numerous users might concurrently try to purchase a specific popular item, as detailed in our Program Example (Section 6.6). In these situations of high competition for specific keys, the explicit locks of pessimistic control offer more predictable behavior by ensuring sequential access to resources. This approach is preferred over potentially numerous optimistic query failures and retries that could occur if OCC were used in such scenarios. While OCC performs well with mostly read operations or broadly distributed writes, the expectation of frequent competition for specific keys makes pessimistic control a more suitable strategy for RedType.

Therefore, RedType will use a Pessimistic Concurrency Control strategy based on explicit locking. This choice favors correctness and manageable complexity for the high contention scenarios we expect. The main challenge with PCC in general is managing deadlocks effectively, but as mentioned, RedType's chosen prevention method aims to eliminate them.

Deadlock Fundamentals

A deadlock occurs when two or more queries are stuck waiting for each other indefinitely. Each query waits for a resource (e.g., a locked key) that another query in the cycle holds. For example, if Query Q1 locks resource A and waits for resource B, while Query Q2 locks B and waits for A, neither can move forward.

For a deadlock to happen, four conditions, known as the Coffman conditions, usually need to be true at the same time [3]. These are:

1. Mutual Exclusion: At least one resource is held in a way that only one query can use it at a time, like an exclusive lock.
2. Hold and Wait: A query holds at least one resource and is waiting to get more resources that other queries are holding.
3. No Preemption: Resources cannot be forcibly taken from queries. A query must release its resources voluntarily.
4. Circular Wait: There is a circular chain of queries. Each query in the chain holds one or more resources that the next query in the chain is waiting for.

Systems can manage deadlocks by preventing them, by detecting and resolving them, or by avoiding them. RedType focuses on deadlock prevention by ensuring that these conditions, particularly circular wait or hold-and-wait leading to cycles, cannot simultaneously arise.

Preventing Deadlocks with Lock Pre-declaration

RedType's chosen strategy requires the pre-declaration of all keys that a query plans to lock during its execution. Before the main query logic starts, the RedType system will try to get exclusive locks on all these specified keys. This method directly attacks the conditions needed for deadlocks. Specifically, by acquiring all necessary locks before execution begins, it prevents a query from holding some locks while waiting for others in a way that could complete a cycle or it effectively imposes an order that prevents circular dependencies from forming [26].

This upfront locking prevents circular waits since locks are acquired systematically at the start. If a query cannot acquire all the necessary locks upfront, it does not begin executing or hold any partial locks. This approach is crucial for preventing deadlocks from occurring.

Furthermore, this strategy is crucial for guaranteeing atomicity for the operations within the query. Once all necessary locks are held, the query runs in complete isolation with respect to those locked keys. It is protected from interference by other concurrent queries attempting to access the same locked keys until it finishes and releases its locks. This makes the script's operations appear as a single indivisible unit regarding the locked data.

While this means developers must identify and declare all required keys beforehand, it offers strong guarantees for atomicity and deadlock prevention. The system must enforce a critical rule: if a query attempts to access a key that was not in its pre-declared lock set in a manner requiring a lock, the execution must stop immediately. Any changes made by the query up to that point ideally should be rolled back to maintain data consistency and uphold the atomicity promise for the operations within the query. This rollback mechanism is vital for data integrity. Implementing a full rollback system would typically involve techniques such as journaling or maintaining undo logs for operations performed within the query before the violation [11, 25]. However, developing such a comprehensive rollback feature for queries is a significant task. For the current project phase, which prioritizes the core language design and type system, the focus will be on detecting such violations and halting execution. The detailed implementation of state rollback for partially executed queries is therefore deferred. This overall design choice puts correctness, predictability, and atomicity first, fitting RedType's core goal of a robust, type safe environment.

6.10 Query Execution Strategy

RedType has opted for server-side interpretation of queries. This approach simplifies client implementation and centralizes schema validation and type safety enforcement on the server, which aligns with RedType's goal of guaranteeing safe database operations. While client-side compilation could theoretically offload work from RedType servers, it would burden each client application with embedding heavy compilation logic for the query language. This model would drastically increase client-side resource use, build complexity for application developers, and maintenance effort across diverse devices.

Within the server-side model, an alternative is server-side compilation, where the server compiles queries into an optimized form (e.g., bytecode). This compiled form can be executed, cached for reuse, and improve performance for common queries. However, for RedType, the chosen approach is direct server-side interpretation. This simplifies server implementation and maintains textual queries as the clear and definitive instructions, aligning with the current focus on the core language and type system. Optimizations like server-side compilation and caching are deferred.

6.11 RedType's Language Architecture

RedType consists of multiple components, being a schema definition, locking declarations and statements.

Schema Definition: The schema definition specifies the key-value data model, including the types, fields, and constraints that govern the structure of data in RedType. This schema file serves as a contract, dictating how data should be structured, validated, and interacted with. By defining the schema separately and enforcing it server-side, RedType ensures data consistency and reduces errors caused by mismatched or undefined data structures.

Locking declaration: Before executing any statements, developers can specify which database keys should be locked. These locks prevents race conditions by protecting shared resources in advance. All keys declared in the schema can be locked, and the lock declarations occur explicitly before any logic is executed.

Statements: The statements in this component, are used to write queries that execute complex logic directly on the RedType server. It allows developers to define reusable functions, manipulate data through various statements and expressions, perform database operations (like `SET`, `GET`, `INCR`, `DEL`), and implement application-specific workflows. The program operates on the data structures defined in the Schema definition, and all operations are type-checked against the schema during server-side interpretation.

With these three core components of the RedType language established, a specific example of a program written in RedType with a useful scenario will be presented. This example

serves to illustrate how the language can be applied in context and will facilitate a clearer understanding of both the abstract and concrete syntax, as introduced in the following sections.

6.12 Summary

This chapter has presented the core design of RedType, focusing on its type safe command set, which ensures that all operations conform to a schema-defined type system. It also introduced structured handling of missing keys through option types, which is essential for avoiding null errors. Concurrency control was addressed via explicit locking, ensuring atomicity during data access and modification. RedType's architecture was outlined, explaining how the components schema definitions, lock declarations, and statements are expressed within RedType. Through a real-world example, we demonstrated how these design elements integrate in practice.

Additionally, the chapter defined the abstract syntax to formalize program structure and introduced a concrete syntax to specify how RedType scripts are written. And lastly, the support for arrays was also included to handle in-memory collections during execution.

With these design principles in place, the next chapter will formalize the behavior of RedType programs through static and dynamic analysis, based on the abstract syntax defined here.

7 Semantics

In this chapter, we define the structural operation semantic (SOS) of RedType, covering both the functional and schema components. SOS specifies how programs executes by describing their behavior through syntax-directed transition rules. These rules are defined in relation to the abstract syntax and guide both static type checking as well as dynamic execution. [12]

The semantics are split into three sections:

1. **Semantic Environments** - The environment that will be used to maintain the states through type checking and execution
2. **Static Semantics -** Enforces type correctness through a type system
3. **Dynamic Semantics -** Describes how the syntactic categories evaluates through execution

7.1 Semantic Environments

RedType's semantics use several environments to maintain state during type checking and execution:

- **Schema Environment** ($\Gamma_s \subseteq \text{Key} \rightharpoonup \text{Type}$)
  Maps database keys to their declared types for static validation.
  - Key: identifiers used in the database
  - Type: statically declared types (`Int`, `Double`, `String`, `Bool`)
- **Type Environment** ($\Gamma \subseteq \text{Var} \to \text{Type}$)
  Tracks types of local variables during type checking.
  - Var: local variable identifiers
  - Type: can be either `Int`, `Double`, `Bool`, `String`, Arrays or `Option<Int|Double|String|Bool>`
- **Variable Environment** ($\mathscr{E} \subseteq \text{Var} \to \text{Loc}$)
  Binds variables to memory locations during execution.
  - Var: variables in scope
  - Loc: the variables memory locations

- **Store** ($\sigma \subseteq \text{Loc} \to \text{Value}$)
  Maintains runtime values of variables at their locations.
  - Loc: memory locations
  - Value: the value at the location (String, Int, Double, Bool, None or Some(v), with v being a value of primitive type)
- **Database** ($DB \subseteq \text{Key} \to \text{Value}$)
  Represents the persistent key-value store.
  - Key: database keys
  - Value: values stored in the database under the key e.g (String, Int, Double or Bool)
- **Function Environment** ($\pi \subseteq \text{FuncName} \to \text{Type}^* \times \text{Type} \times \text{Stmt} \times \text{Var}^* \times \mathscr{E} \times \pi$)
  Records full function definitions, including typing and runtime information:
  - FuncName: function identifier
  - Type^*: list of parameter types (String, Int, Double, Bool, Array)
  - Type: return type (String, Int, Double, Bool, Array)
  - Stmt: function body
  - Var^*: parameter names
  - $\mathscr{E}$: environment at the time of function definition
  - π: function environment at definition time
- **Lock Environment** ($\mathscr{L} \subseteq \text{Key} \to \mathit{Bool}$)
  This environment indicated whether a key is locked (true) or not (false).
  - Key: identifiers of locked database keys
  - Bool: true or false depending on the keys state

Furthermore there are some general terms which will be used:

- $\vdash_l$: the l is the next possible place to store a variable in memory, meaning $\vdash_{l+1}$ is the one thereafter.

For scope handling the following helper functions will be used. These functions have been defined by Hüttel in the GitHub repository *The Dims language processor* [42].

- **Enter Scope**: $\texttt{enter}(\mathscr{E}) = \mathscr{E}'$,where $\mathscr{E}'$ is identical to $\mathscr{E}$ except there has been pushed a new scope on top.
- **Leave Scope**: $\texttt{leave}(\mathscr{E}) = \mathscr{E}'$, where $\mathscr{E}'$ is identical to $\mathscr{E}$ except the top scope has been removed.

Transition system

For the semantics, it is helpful to define a transition system. A transition system consists of configurations, representing the current state of the program, and transitions, representing the execution steps. If a configuration has no further transitions, it is considered a terminal configuration. [12]

The transition system is defined as a 3-tuple:

$$(\mathcal{C}, \rightarrow, T)$$

Where,

- $\mathcal{C}$ represents the set of possible configurations.
- $\rightarrow$ represents the transition relation, which is a subset of T.
- $T \subseteq \mathcal{C}$ represents the set of terminal configurations. [12]

The transition system for RedType is detailed in Appendix D.

In the following subsections the semantic rules for the type system, statements, schema, and database operations will be covered. In order to keep the sections concise only a select few semantic rules will be shown, while other will still be explained, they can only be seen in appendix C.

7.2 Static Semantics - Type System

Type checking will be performed statically before execution. This allows the RedType to reject a program that would otherwise produce run-time errors such as applying arithmetic operation to boolean values, thereby ensuring a well-typed language.

Function Definition and Call

Functions in RedType are statically type-checked to ensure correctness prior to execution. The type system enforces that each function is declared with a well-typed signature, including the parameter types and return type, and that the function body adheres to this specification.

$$[\text{FUNC-DECL}_{TS}] \quad \frac{\Gamma[x_1 \mapsto T_1, \ldots, x_n \mapsto T_n] \;\vdash\; S : T_{\text{ret}}}{\Gamma, \pi \;\vdash\; \texttt{func}\ f(x_1 : T_1, \ldots, x_n : T_n) : T_{\text{ret}}\{S\} : \texttt{ok} \;\Rightarrow\; \pi' = \pi\big[f \mapsto \big((T_1, \ldots, T_n),\, T_{\text{ret}},\, S\big)\big]}$$

$$[\text{FUNC-CALL}_{TS}] \quad \frac{\pi(f) = \big((T_1, \ldots, T_n),\, T_{\text{ret}},\, S\big) \quad \Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n}{\Gamma, \pi \;\vdash\; f(e_1, \ldots, e_n) : \texttt{ok}}$$

The typing rule [FUNC-DECL$_{\text{TS}}$] describes how function declarations are validated. Given a function `f` with parameters $x_1 : T_1, \ldots, x_n : T_n$, and a return type T_{ret}, the function body S must type-check under an extended type environment where each parameter is bound to its corresponding type. If the body is well-typed, the function is added to the function environment π with its full signature and definition.

The rule [FUNC-CALL$_{\text{TS}}$] ensures that function calls are valid with respect to the function's type signature. To type-check a call to a function $f(e_1, \ldots, e_n)$, the function environment π must contain an entry for f with the parameter types $(T_1, \ldots, T_n)$. Each actual argument e_i must type-check to the corresponding expected type T_i. If all arguments match their expected types, the function call is accepted as well-typed.

These rules guarantee that functions only are defined if their body respects the declared return type, only are called with the correct number and type of arguments, and Runtime type errors due to function misuse are eliminated by static analysis.

Statements

These rules define how various types of assignment statements are checked in a statically typed setting.

$$[\text{VAR-DECL-INIT}_{TS}] \quad \frac{\Gamma' = \Gamma[x \mapsto T]\ \Gamma \vdash e : T}{\Gamma \vdash x : T = e : \texttt{ok}}$$

$$[\text{ARRAY-ASSIGN}_{TS}] \quad \frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = T[] \quad \Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash x[e_1] = e_2 : \texttt{ok}}$$

A variable declaration with initialization is valid if the initializer expression matches the declared type, after which the variable is added to the typing environment. Simple assignments require that the variable is already declared and that the assigned expression matches its type (checked using the 'dom' helper function). For array assignments, the variable must be of an array type, the index must be an integer, and the value being assigned must match the array's element type.

$$[\text{IF-THEN-ELSE}_{TS}] \quad \frac{\Gamma \vdash b : \texttt{bool} \quad \texttt{enter}(\Gamma) \vdash S_1 : \texttt{ok} \quad \texttt{enter}(\Gamma) \vdash S_2 : \texttt{ok}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{ok}}$$

$$[\text{FOR}_{TS}] \quad \frac{\Gamma \vdash e : T[] \quad \texttt{enter}(\Gamma)[x \mapsto T] \vdash S : \texttt{ok}}{\Gamma \vdash \texttt{for}\ x\ \texttt{in}\ e\ S : \texttt{ok}}$$

Conditional and loop constructs enforce that the guard expression has type `bool`. In `if-then` and `if-then-else`, the bodies of the branches are type-checked in a new local scope derived from the current environment. The `for`-loop iterates over an array, introducing a new variable bound to the array element type within the loop body. Similarly, the `while`-loop checks that its body is well-typed under the same local scope.

$$[\text{RETURN-EXPR}_{TS}] \quad \frac{\Gamma \vdash e : T_{ret}, \text{ where } T_{ret} = \text{return type of the enclosing function}}{\Gamma, \pi \vdash \texttt{return}\ e : \texttt{ok}}$$

These rules cover control flow statements. The `skip` statement is always considered well-typed as it performs no action. Return statements are validated based on the expected return type of the enclosing function. If the function specifies a return type, then the returned expression must match it. If no return type is declared, a bare `return` is allowed. In both cases, the typing environment is extended with return context information.

$$[\text{MATCH}_{TS}] \quad \frac{\begin{array}{l}\Gamma \vdash e : \texttt{Option}\langle T \rangle \\ \forall i \in 1..n,\ \big(\text{if } Case_i = \texttt{Some}(x_i) \Rightarrow S_i,\ \Gamma[x_i \mapsto T] \vdash S_i : \texttt{ok} \\ \quad \wedge\ \text{if } Case_i = \texttt{None} \Rightarrow S_i,\ \Gamma \vdash S_i : \texttt{ok}\big)\end{array}}{\Gamma \vdash \texttt{match}\ e\ \{\texttt{Case}_i\}_{i=1}^{n} : \texttt{ok}}$$

The matching construct allows for a convenient way to handle optional values by pattern matching. The [MATCH_{TS}] rule allows for multiple match branches, each being either a `Some` or `None` case. For `Some`(x), the variable is added to the environment with type T, and the body must type-check accordingly. For `None`, the body is checked under the existing environment. This ensures that all branches of a match expression are safely and consistently type-checked, supporting robust handling of optional values in the language.

$$[\text{STMT-SEQ}_{TS}] \quad \frac{\forall i \in [1..n],\ \Gamma_{i-1}, \sigma_{i-1} \vdash S_i : \texttt{ok} \Rightarrow (\Gamma_i, \sigma_i)}{\Gamma_0, \sigma_0 \vdash \overrightarrow{S} : \texttt{ok} \Rightarrow (\Gamma_n, \sigma_n)} \quad \text{where } \overrightarrow{S} = S_1, S_2, \ldots, S_n$$

[$\text{STMT-SEQ}_{\text{TS}}$] defines how statement sequences are type-checked. It iterates over all statements, and if all of them are well typed, then their sequential composition is also well typed. This rule forms the basis for building complex behaviors from simpler statements, ensuring that each part of the sequence maintains type safety.

Database Operations

Database operations in RedType interact with key-value stores and must be type-checked to ensure correctness and consistency with the declared schema. The type system prevents illegal operations, such as incrementing a Bool or adding a string to a numeric set by statically validating all database commands against the schema environment Γ_s.

As mentioned before, Γ_s maps database keys to their declared types. The following typing rules ensure that the database commands adheres to the expected type constraints.

The increment rules describe how increment operations are type checked:

$$[\text{INCR}_{TS}] \quad \frac{\forall K_i \in \vec{K},\ \Gamma_s(K_i) \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma_s \vdash \texttt{INCR}\ \vec{K} : \texttt{ok}}$$

The [INCR_{TS}] rule states that increment operations can only be applied to keys that are declared as either integers or double in the schema environment.

$$[\text{INCRBY-DOUBLE}_{TS}] \quad \frac{\Gamma_s, \sigma \vdash e \Rightarrow_{Exp} v \quad v \Rightarrow_{TS} \{\texttt{Int}, \texttt{Double}\} \quad \forall K \in \vec{K},\ \Gamma_s(K_i) \in \{\texttt{Double}\}}{\Gamma_s, \sigma \vdash \texttt{INCR}\ \vec{K}\ \texttt{BY}\ e : \texttt{ok}}$$

The [$\text{INCRBY}_{\text{TS}}$] rules state that all keys in the sequence $\vec{K}$ must be declared as either an integer or double in the schema environment. If K is an Int then e must evaluate to an Int, but if K is a Double, then e can evaluate to either an Int or Double. The same logic applies to the DECR and DECRBY commands.

The following database operations describes how a SET and GET are type checked:

$$[\text{SET}_{TS}] \quad \frac{\Gamma_s(K) = T \quad e \Rightarrow_{TS} T}{\Gamma_s \vdash \texttt{SET}\ K = e : \texttt{ok}}$$

The SET operation is valid if the key K is declared in the schema environment Γ_s with type T and the expression e being assigned is also type T. This makes sure that updates to the database is consistent with the declared types.

Since RedType relies on type safety, the GET operation must be handled using an Option type. When retrieving a value with GET, there is no guarantee that the key exists in the database. Therefore the result must be treated as an optional value to account for this possibility as shown below. On the other hand if a variable has already been used for a GET, it thereby has the type of Option<T>, and can be reassigned with a new GET:

$$[\text{GET}_{TS}] \quad \frac{\Gamma_s(K) = T}{\Gamma_s,\ \Gamma \vdash x : \texttt{Option<}T\texttt{>} = \texttt{GET}\ K : \texttt{ok} \quad \text{and} \quad \Gamma' = \Gamma[x \mapsto \texttt{Option<}T\texttt{>}]}$$

This rule states that if the key K is declared in the schema environment Γ_s with the type T, then performing a GET operation on K this will result in x being assigned the Option<T> type.

$$[\text{DELETE}_{TS}] \quad \frac{\forall K_i \in \vec{K},\ K_i \in \text{dom}(\Gamma_s)}{\Gamma_s \vdash \texttt{DEL}\ \ \vec{K} : \texttt{ok}}$$

The [$\text{DELETE}_{\text{TS}}$] rule makes sure that each key in the sequence $\vec{K}$ exists in the schema environment Γ_s. This guarantees that deletions are only performed on declared keys.

7.3 Dynamic Semantics

In this section, the dynamic semantics of RedType are defined. This section mirrors the structure of the previous Type System section, with the addition of the Schema definition rules. A transition system will be introduced, used to define the structural operational semantics (SOS) using a combination of big-step and small-step semantics.

Big-step and small-step semantics

Structural operational semantics can be done in two ways, the first being big step semantics, which evaluate entire computations in a single step. It describes the overall evaluation of a program starting in a configuration γ and producing a final, terminal configuration γ', written as $\gamma \to \gamma'$. The second is small-step semantics, which break down computation into a sequence of individual transitions. Each step transforms a configuration γ into a new configuration γ' written as $\gamma \Rightarrow \gamma'$ where γ' is not necessarily terminal configuration. The full evaluation is described as a sequence of such transitions. [12]

Since both big-step and small-step semantics have their respective advantages and drawbacks depending on the context, RedType will make use of both. For simpler constructs where the intermediate computation steps are not of interest, big-step semantics will be used, as they are easier to formulate. However, for cases where it is important to observe the intermediate transitions such as database operations, small-step semantics will be applied.

Program

RedType program consists of three components, a schema declarations which must either be a single schema definition or no schema definition if one already exists. A sequence of lock declarations, and a sequence of statements.

$$[\text{PROGRAM}_{SSS}] \quad \frac{\Gamma_s \vdash \overrightarrow{S} \Rightarrow_{Schema} \Gamma'_s \quad \mathscr{L} \vdash \overrightarrow{L} \to \mathscr{L}' \quad \langle \overrightarrow{S_t}, \sigma, \mathscr{E}, \mathscr{L}', DB \rangle \Rightarrow (\sigma', \mathscr{E}', DB')}{\langle \overrightarrow{S}\ \ \overrightarrow{L}\ \ \overrightarrow{S_t}, \sigma, \mathscr{E}, \mathscr{L}, DB, \Gamma \rangle \Rightarrow_{Prog} (\sigma', \mathscr{E}', \mathscr{L}, DB', \Gamma')}$$

The rule states that the schema declaration $\vec{S}$ is evaluated in the initial schema environment producing an updated environment. Although, the syntax permits the sequence $\vec{S}$, the semantic enforces only at most one schema to ensure schema uniqueness. The lock declaration $\vec{L}$ are evaluated, producing a new lock environment $\mathscr{L}'$. Finally the sequence of statements are evaluated with the current store environment, variable environment, updated lock environment and current database resulting in a final configuration.

Locks

The lock declaration rules define how a sequence of keys is processed into the lock environment $\mathscr{L}$, which tracks which keys are currently locked. The rule describes how the environment is updated as each key is defined in a lock declaration. The notation '::' defines a head (left side) and tail (right side), essentially just representing an array of Keys.

$$[\text{LOCK-DECL}_{SSS}] \quad \frac{\mathscr{L}(K_1) \neq \texttt{true} \quad \mathscr{L}' = \mathscr{L}[K_1 \mapsto \texttt{true}]}{\mathscr{L} \vdash \texttt{LOCK}\ \ K_1 :: K_{rest} \Rightarrow_{Lock} \langle \texttt{LOCK}\ K_{rest}, \mathscr{L}' \rangle}$$

LOCK-DECL defines a recursive case where a lock is acquired on a K_1 provided it is not already locked in L. The environment is extended with a binding from K_1 to a boolean value *true*, indicating the key is now locked. All locks in a program will be release at the end of it (as seen in [PROGRAM$_{SSS}$])

Schema

In RedType, schemas are defined as mappings from field names (keys) to their corresponding types. The schema environment, denoted by Γ_s, maintains associations between schema names and their respective field-to-type mappings. The semantics of schemas are expressed using big-step transitions, which are straightforward due to the simplicity of schemas, as they involve only direct assignments of types to field names.

Notation:

- Γ_s is as mentioned earlier, the schema environment.
- Σ is a finite map from field names to types (e.g., $\{\texttt{age} \mapsto \texttt{int}, \texttt{name} \mapsto \texttt{string}\}$).
- $k, k_1, \ldots, k_n$ are field names.
- $T, T_1, \ldots, T_n$ are types.

[SCHEMA-FIELD$_{BSS}$] A schema containing a single field maps the field name to its type.

$$\frac{}{\Gamma_s \vdash \{k : T\} \Rightarrow_{Schema} \{k \mapsto T\}}$$

[SCHEMA-MULTI$_{BSS}$] Multiple fields can be evaluated recursively by creating a union between individual field mappings.

$$\frac{\Gamma_s \vdash F \Rightarrow \Sigma_1 \quad \Gamma_s \vdash \{k : T\} \Rightarrow \Sigma_2}{\Gamma_s \vdash F \cup \{k : T\} \Rightarrow_{Schema} \Sigma_1 \cup \Sigma_2}$$

[SCHEMA-DEF$_{BSS}$] A full schema definition binds a name to the evaluated field map in the schema environment.

$$\frac{\Sigma = \Gamma_s \vdash \{x_1 : T_1, \ldots, x_n : T_n\}}{\Gamma_s \vdash \mathtt{x}\{x_1 : T_1, \ldots, x_n : T_n\} \Rightarrow_{Schema} \Gamma_s[\mathtt{x} \mapsto \Sigma]}$$

Function Definition and Call

Function definitions and function calls are handled using small-step semantics. The definition of a function updates the function environment, while a call executes the body of the function in a new environment. The following rules describe this behavior.

$$\text{[FUNC-DEF}_{SSS}\text{]} \quad \frac{\pi' = \pi\big[f \mapsto \big((T_1, \ldots, T_n),\, T_{\text{ret}},\, S,\, [x_1, \ldots, x_n],\, \mathscr{E},\, \pi\big)\big]}{\mathscr{E}, \pi \;\vdash_l\; \big\langle\; \texttt{func}\;\; f(x_1 : T_1, \ldots, x_n : T_n) : T_{\text{ret}}\{S\},\; \sigma \big\rangle \;\rightarrow\; \langle \sigma, \pi' \rangle}$$

$$\text{[FUNC-CALL}_{SSS}\text{]} \quad \frac{\begin{array}{l} \pi(f) = \big(S,\, [x_1, \ldots, x_n],\, \mathscr{E}',\, \pi'\big), \\ \forall i \in \{1..n\}.\, \mathscr{E}(y_i) = \ell_i,\; \sigma(\ell_i) = v_i, \\ \text{fresh locations } \ell_i' \text{ for each } i \text{ where } v_i \text{ is not an array}, \\ \mathscr{E}'' = \mathscr{E}'\left[x_i \mapsto \begin{cases} \ell_i & \text{if } v_i \text{ is an array} \\ \ell_i' & \text{otherwise} \end{cases} \;\middle|\; i \in \{1..n\}\right], \\ \sigma' = \sigma\left[\ell_i' \mapsto v_i \;\middle|\; v_i \text{ is not an array}\right], \\ \pi'' = \pi'[f \mapsto (S, [x_1, \ldots, x_n], \mathscr{E}', \pi')], \\ \mathscr{E}'', \pi'' \;\vdash_l\; \langle S, \sigma' \rangle \rightarrow \sigma'' \end{array}}{\mathscr{E}, \pi \;\vdash_l\; \langle f(y_1, \ldots, y_n), \sigma \rangle \rightarrow \sigma''}$$

The rule above, [FUNC-DEF$_{SSS}$], defines how a function declaration is evaluated. When a function `f` is declared with parameters $(x_1, \ldots, x_n)$ and body S, it is stored in the function environment π as a closure. This closure captures the current environment $\mathscr{E}$ and the current function environment π, allowing for lexical scoping and recursion. The store σ remains unchanged since function definition does not perform any computation.

The actual behavior of parameter passing, whether by value or by reference, is determined at the point of function call, according to the [FUNC-CALL$_{SSS}$] rule. In this semantics, arrays are passed by reference, meaning the callee accesses the same memory as the caller, while all other types are passed by value, meaning their values are copied into fresh locations within the function's environment.

Statements

We define the semantics of statements using big-step rules. These describe how complete statements transform the current store and environment. Each rule below corresponds to a different form of statement in the language.

A variable declaration evaluates the expression and binds the resulting value to a fresh location, which is recorded in the extended environment. Additionally the assignment rule evaluates the expression on the right-hand side, looks up the location of the left-hand variable in the environment, and updates the store accordingly.

$$[\text{DEC}_{SSS}] \quad \frac{x \notin \texttt{dom}(\mathscr{E}) \quad \sigma \circ \mathscr{E} \vdash e \ \rightarrow_{\text{Exp}} v \quad \mathscr{E}' = \mathscr{E}[x \mapsto l]}{\mathscr{E} \vdash_l \langle x : T = e, \sigma, DB\rangle \Rightarrow_S (\sigma[l \mapsto v], \mathscr{E}', DB)}$$

The `skip` statement has no effect; it simply returns the current state unchanged.

$$[\text{SKIP}_{SSS}] \quad \frac{}{\mathscr{E} \vdash_l \langle \text{skip}, \sigma, DB\rangle \Rightarrow_S (\sigma, DB)}$$

Sequencing of statements is handled in two parts. If the first statement in a sequence takes a step to an intermediate statement, this reduction is propagated through the sequence using the first rule. Otherwise, if the first statement completes execution (i.e., yields a final store and database), the sequence proceeds by continuing with the next statement. This behavior is captured by the [SEQ-STEP$_{SSS}$] and [SEQ-DONE$_{SSS}$] rules.

$$[\text{SEQ-STEP}_{SSS}] \quad \frac{\mathscr{E} \vdash_l \langle S_1, \sigma, DB\rangle \Rightarrow_S \langle S_1', \sigma', DB'\rangle}{\mathscr{E} \vdash_l \langle S_1 :: S_{rest}, \sigma, DB\rangle \Rightarrow_s \langle S_1' :: S_{rest}, \sigma', DB'\rangle}$$

$$[\text{SEQ-DONE}_{SSS}] \quad \frac{\mathscr{E} \vdash_l \langle S_1, \sigma, DB\rangle \Rightarrow (\sigma', DB')}{\mathscr{E} \vdash_l \langle S_1 :: S_{rest}, \sigma, DB\rangle \Rightarrow \langle S_{rest}, \sigma', DB'\rangle}$$

$$\sigma \circ \mathscr{E} \vdash e \ \rightarrow_{\text{Exp}} v$$

In the production rules for the following control structure statements, the expression e must evaluate to a boolean value. This is necessary because the syntactic category `Exp` contains both arithmetic and boolean expressions:

Conditional statements evaluate the guard expression. If it evaluates to `tt` (true), the `then` branch is executed; if `ff` (false), the `else` branch (if present) is executed instead. The semantics ensure we only enter the chosen branch. Here only the 'if-else-true' version is shown, however the 'if-else-false', 'if-true', and 'if-false' follow the same logic.

$$[\text{IF-ELSE-TRUE}_{SSS}] \quad \frac{\begin{array}{c}\sigma \circ \mathscr{E} \vdash e \ \rightarrow_{\text{Exp}} \texttt{tt} \quad \mathscr{E}' = \texttt{enter}(\mathscr{E}) \\ \mathscr{E}' \vdash_l \langle S_1, \sigma, DB\rangle \Rightarrow_S (\sigma', DB')\texttt{leave}(\mathscr{E}')\end{array}}{\mathscr{E} \vdash_l \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma, DB\rangle \Rightarrow_S (\sigma', DB')}$$

`While` loops repeatedly execute their body as long as the condition evaluates to `tt`. Otherwise, they terminate immediately.

$$[\text{WHILE-TRUE}_{SSS}] \quad \frac{\begin{array}{c}\sigma \circ \mathscr{E} \vdash e \rightarrow_{\text{Exp}} \texttt{tt} \quad \mathscr{E}' = \texttt{enter}(\mathscr{E}) \\ \mathscr{E}' \vdash_l \langle S, \sigma, DB\rangle \Rightarrow_S (\sigma', DB') \quad \mathscr{E}'' = \texttt{leave}(\mathscr{E}')\end{array}}{\mathscr{E} \vdash_l \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma, DB\rangle \Rightarrow_S \langle S; \texttt{while}\ e\ \texttt{do}\ S, \sigma', DB'\rangle}$$

$$[\text{WHILE-FALSE}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \ \rightarrow_{\text{Exp}} \texttt{ff}}{\mathscr{E} \vdash_l \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma, DB\rangle \Rightarrow_S (\sigma, DB)}$$

Finally, return statements either yield a value or not. When an expression is provided, it is evaluated and returned along with the current state.

$$[\text{RET-VAL}_{\text{SSS}}] \quad \frac{\sigma \circ \mathscr{E} \ \vdash e \ \rightarrow_{\text{Exp}} \ v}{\mathscr{E} \ \vdash_l \ \langle \texttt{return}\ e,\ \sigma,\ DB\rangle \ \Rightarrow_S \ (\sigma,\ DB,\ v)}$$

$$[\text{RET}_{\text{SSS}}] \quad \frac{}{\mathscr{E} \ \vdash_l \ \langle \texttt{return},\ \sigma,\ DB\rangle \ \Rightarrow_S \ (\sigma,\ DB)}$$

$$[\text{MATCH-SOME}_{SSS}] \quad \frac{\begin{array}{c}\mathscr{E}(x) = l_x \quad \sigma(l_x) = \texttt{Some}(v) \\ \mathscr{E}' = \mathscr{E}[y \mapsto l_1] \quad \sigma' = \sigma[l_1 \mapsto v] \\ \sigma', \mathscr{E}' \vdash_{l_2} \langle S_1, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}, DB'\rangle\end{array}}{\sigma, \mathscr{E} \vdash_{l_1} \langle \texttt{match}\ x\ \texttt{with Some}(y)\ S_1\ \texttt{|}\ \texttt{None}\ S_2, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}'', DB'\rangle}$$

The [MATCH-SOME$_{SSS}$] rule applies when the matched expression x evaluates to `Some`(v). The environment maps x to a location l_x, and the store maps l_x to the value `Some`(v).

A fresh location l_1 is introduced and bound to y, storing the value v at that location. The statement S_1 is evaluated under the extended environment and updated store. After evaluation, the original environment $\mathscr{E}$ is restored, ensuring that the binding of y is local to the match arm. The [MATCH-NONE$_{SSS}$] rule applies when the matched expression x evaluates to `None`, and then S_2 is evaluated under the original environment and store. In this case, no new bindings are introduced, other than that, the logic is the same.

Database Operations

We define the semantics of database-related operations using small-step rules. These operations manipulate the key-value store DB and reflect how each statement modifies it, often alongside the program state s.

The `DEL` operation processes a sequence of keys. For each key, it updates the database by removing the corresponding entry, represented here by setting its value to `null`. Once all keys have been processed, the operation reduces to `skip`.

A helper function for deleting keys from the DB environment has been defined:

$$\text{delete(DB, K) = DB}' \quad \text{such that} \quad \text{DB}'(\text{k}) = \begin{cases} DB(k) & \text{if } k \neq K \\ \text{undefined} & \text{if } k = K \end{cases}$$

$$\text{[DEL-STEP}_{SSS}\text{]} \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB' = \text{delete}(DB, K)}{\langle \texttt{DEL}\ \ K :: K_{\text{rest}},\ DB,\ \sigma,\ \mathscr{L} \rangle \ \Rightarrow_{Db}\ \langle \texttt{DEL}\ \ K_{\text{rest}},\ DB',\ \sigma,\ \mathscr{L} \rangle}$$

$$\text{[DEL-DONE}_{SSS}\text{]} \quad \frac{}{\langle \texttt{DEL}\ \ [\,],\ DB,\ \sigma,\ \mathscr{L} \rangle \ \Rightarrow\ \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L} \rangle}$$

The increment operation applies to a list of keys. Each key is incremented one at a time, in sequence, by increasing its associated value in the database. Once all keys have been processed, the operation completes.

$$\text{[INCR-STEP}_{SSS}\text{]} \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) + 1]}{\langle \texttt{INCR}\ \ K :: K_{\text{rest}},\ DB,\ \sigma,\ \mathscr{L} \rangle \Rightarrow_{Db} \langle \texttt{INCR}\ \ K_{\text{rest}},\ DB',\ \sigma,\ \mathscr{L} \rangle}$$

$$\text{[INCR-DONE}_{SSS}\text{]} \quad \frac{}{\langle \texttt{INCR}\ \ [\,],\ DB,\ \sigma,\ \mathscr{L} \rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L} \rangle}$$

Similarly, The decrement operation processes a list of keys one at a time, subtracting one from the value associated with each key in the database. Once all keys have been processed, the operation completes.

The INCR BY and DECR BY commands generalize the increment and decrement operations to allow for arbitrary values. These values are evaluated once, and the resulting number is then added to or subtracted from the corresponding database entry for each specified key. Similarly to INCRBY, DECRBY follows the same logic.

$$[\text{INCRBY-STEP}_{SSS}] \quad \frac{\sigma \vdash e \to_{\text{Aexp}} v \quad \mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) + v]}{\langle \texttt{INCR}\ K :: K_{\text{rest}}\ \texttt{BY}\ e, DB, \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{INCR}\ K_{\text{rest}}\ \texttt{BY}\ e, DB', \sigma, \mathscr{L}\rangle}$$

$$[\text{INCRBY-DONE}_{SSS}] \quad \frac{}{\langle \texttt{INCR}\ [\,]\ \texttt{BY}\ e, DB, \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip}, DB, \sigma, \mathscr{L}\rangle}$$

The 'SET' operation assigns a new value to a key in the database. First, the expression is reduced to a value; then the key is updated.

$$[\text{SET}_{SSS}] \quad \frac{\sigma \vdash e \to_{\text{Aexp}} v \quad DB' = DB[K \mapsto v]}{\langle \texttt{SET}\ K\ \texttt{TO}\ e, DB, \sigma\rangle \Rightarrow \langle \texttt{skip}, DB', \sigma\rangle}$$

The 'GET' operation fetches a value from the database and stores it in a variable in the state. Since the result of the GET operation needs to be evaluated using a match case, then the value of x is either Some(v) or None, depending on the keys existence. There are both rules for a typed GET and a non-typed GET (e.g x = GET K), but only the typed versions are shown here.

$$[\text{GET-TYPED}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB(K) = v}{\langle x : \texttt{Option}\langle T\rangle = \texttt{GET}\ K, DB, \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip}, DB, \sigma[x \mapsto \texttt{Some}(v)]\rangle}$$

$$[\text{GET-TYPED-NONE}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad K \notin \text{dom}(DB)}{\langle x : \texttt{Option}\langle T\rangle = \texttt{GET}\ K, DB, \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip}, DB, \sigma[x \mapsto \texttt{None}]\rangle}$$

This chapter has established the formal semantics of RedType, providing a clear framework for both type checking and execution. By introducing semantic environments, a static type system, and dynamic evaluation rules, we ensure that RedType programs can be both well-typed and predictably executed. These structural operational semantics (SOS) serve as a foundational guide for understanding how RedType behaves, ensuring consistency between the schema and functional components of the language.

8 Technology

This section presents the core technologies and decisions shaping RedType's implementation. It elaborates on the choice of programming language and parser generation approach, including an overview of the alternatives evaluated, to meet project requirements for robust type handling and overall development efficiency.

8.1 Rust

Rust is a high-performance, safety-focused programming language that is designed to address challenges found in other languages, like C and C++. These challenges, mainly revolve around memory errors and complexities when it comes to concurrent programming. The effectiveness of Rust has led major tech companies such as Microsoft, Google, Facebook, and Discord to adopt it for operating systems, browser engines, web services, and other system-level applications. [10]

Rust provides a unique combination of speed, memory safety, and concurrency support without forcing developers to choose between them. In contradiction to other system languages, Rust ensures safety at compile time, thereby eliminating entire classes of runtime errors while maintaining high performance. Rust is a solid choice when it comes to developing a new programming language since it has a lot of key advantages when it comes to language development, including memory safety without garbage collection, high-performance native execution, strong concurrency support, robust tooling, and cross-platform capabilities. [50]

The combination of speed, safety, and modern tooling Rust offers makes it a compelling choice for implementing the RedType interpreter. Its memory safety and performance allow RedType to run efficiently without the need for manual memory management. Rust's robust concurrency model supports parallel execution, which is crucial for efficiency when interpreting complex code. With tools like Cargo and Rust Analyzer, development of the interpreter is enhanced, while Rust's cross-platform support ensures RedType can be deployed seamlessly across different operating systems. In summary, Rust provides the ideal technical foundation for building a secure, high-performance RedType interpreter. [10, 17, 50]

Tokio

Tokio is an asynchronous runtime library for the Rust programming language. It provides the building blocks needed for writing network applications, offering a foundation for non-blocking operations and enabling high levels of concurrency with minimal overhead. Asynchronous programming allows a program to continue executing other tasks while waiting for slow operations (like network requests or file reads) to complete, rather than

stopping and waiting. It is built upon Rust's `async/await` features and is the most widely used standard for asynchronous programming in the Rust ecosystem. [44]

The Tokio Runtime: At its core, Tokio provides a multithreaded, work-stealing scheduler, which is a system that automatically redistributes work between processors to maintain efficiency. In this system, idle threads take pending tasks from busier ones to balance the load. This runtime efficiently distributes asynchronous tasks across its pool of threads. It allows developers to write asynchronous code that looks similar to synchronous code but runs non-blockingly, improving resource utilization by not wasting time waiting for operations to complete. [44]

Tasks and Concurrency: Tokio enables concurrency with the ability to handle multiple operations at the same time through lightweight, non-blocking tasks, often referred to as "green threads". Green threads are virtual threads managed by the runtime rather than the operating system, making them much more efficient to create and switch between. These tasks yield control back to the runtime when they encounter an operation that would block. This model allows a small number of operating system threads to handle a very large number of concurrent operations, making it highly scalable. [44]

Multithreading - OS Threads & Worker Threads: Tokio's multithreaded runtime leverages a fixed-size pool of OS threads (managed by the operating system), each acting as a worker thread that executes asynchronous tasks. These workers manage tasks via local queues and employ a work-stealing strategy to balance load, enabling cooperative scheduling where tasks yield control, allowing thousands of async tasks to be multiplexed onto the OS threads. This minimizes costly context switches inherent in multitasking, thereby maximizing system throughput. [44]

Asynchronous Networking: Tokio provides asynchronous APIs for common networking protocols like TCP, used for reliable data transmission and UDP, used for fast, connectionless communication. These APIs integrate seamlessly with the runtime, allowing applications to handle many network connections concurrently without dedicating a thread to each one. This is crucial for building performant network services or language features involving network communication. [44]

Synchronization Primitives: When multiple asynchronous tasks need to access shared data or send information to each other, using traditional synchronization mechanisms can cause issues. These older tools often "block" an entire OS thread if a resource isn't immediately available, meaning that the thread can't do any other work. Tokio offers its own set of asynchronous synchronization primitives, like tokio::sync::Mutex (to ensure only one task accesses a piece of data at a time) and various types of channels (for structured communication between tasks). The key difference is that when a Tokio task encounters one of these primitives and has to wait (e.g., for a Mutex to be released or a message to arrive), it

doesn't stall the OS thread. Instead, the task yields control, effectively pausing itself and allowing the runtime to use that OS thread to execute other ready tasks, thus maintaining system responsiveness and high concurrency. [44]

8.2 Choice of Parser Generator

The selection of an appropriate parsing strategy and the corresponding generator tool is an important step in implementing a compiler or interpreter. For the RedType language, developed in Rust, this choice impacts development efficiency, maintainability, and the robustness of the resulting parser. Several parsing methodologies and Rust-based tools were evaluated.

What is a Parser?

A parser is a part of a program (often a compiler or interpreter) that takes raw text (e.g user written code), and turns it into something the computer can understand and work with. Before the parser can do its job, the text needs to be cleaned up and broken into smaller pieces, which is where the lexer takes part. [22]

The lexer (also known as scanner or tokenizer) takes the raw input text and splits it into tokens. Tokens are small, meaningful chunks of the text, like keywords (`if`, `while`), identifiers (`x`, `userName`), operators (`+`, `==`), or punctuation (`{`, `;`). This process is called *tokenizing*. [22]

To recognize these tokens, the lexer uses *regular expressions*. Regular expressions are patterns that describe what certain types of text look like, for example what a number looks like, or how to recognize a valid variable name. For instance, the regular expression `[0-9]+` matches all integers, so it would recognize numbers like `5`, `42`, or `123`. Similarly, the pattern `[a-zA-Z_][a-zA-Z0-9_]*` matches all strings built of small/large charactes and numbers. These patterns help the lexer break down code like `total = 42` into individual tokens such as `IDENTIFIER("total")`, `EQUALS("=")`, and `NUMBER("42")`. [22]

Once the input has been split into a stream of tokens, this stream is passed to the parser. The parser uses a set of grammar rules, e.g. Context-Free Grammars (CFGs) or Parsing Expression Grammars (PEGs), that define how tokens can be combined to form valid structures in the language. [22]

The parser organizes these tokens based on the grammar and builds a structured representation of the code, called an abstract syntax tree (AST). This tree captures the logical structure of the program, which can then be further processed, interpreted, or compiled into executable code. [22]

Different Parsing Strategies

Parsing source code can be approached using several distinct strategies. This section provides a brief overview of the primary methods evaluated for RedType:

Parser Expression Grammars (PEG)

PEGs represent a formalism distinct from traditional Context-Free Grammars (CFGs). They employ a top-down parsing approach characterized by prioritized choices and unlimited lookahead, often combined with memoization (packrat parsing). PEGs inherently avoid ambiguity, always yielding a single parse tree if the input is valid according to the grammar. However, these theoretical differences from the CFG to concepts familiar for the team, led to the decision to pursue CFG-based parsing strategies instead. [47]

Top-Down Parsing (LL)

LL parsers, particularly LL(1), operate top-down, typically using recursive descent with a single token of lookahead. They are relatively straightforward to implement manually and parse input in linear time. However, they impose constraints on the grammar, struggling with left recursion and common prefix issues, often necessitating grammar transformations like left-factoring. [46]

Bottom-Up Parsing (LR)

LR parsers analyze input from left to right and construct a rightmost derivation in reverse (hence LR). This bottom-up approach builds the parse tree from the leaves (tokens) up to the root (start symbol). LR parsers utilize a shift-reduce strategy with a stack to hold grammar symbols and a buffer for incoming tokens. A shift action results in the next input token being pushed onto the stack. Instead, a reduce action occurs when a sequence of symbols on top of the stack matches a grammar production. The symbols are then replaced by the non-terminal from the grammar rule, thereby reducing the input into a higher level of the syntax. When the stack is empty and there is no incoming tokens, the input is parsed correctly and accepted. If it is not possible to shift or reduce, a syntax error is thrown. For instance, if the parser finds multiple production rules in a reduce situation, the syntax must be ambiguous, and an error is thrown. [28]

LR parsers can handle a broad class of grammars, including left-recursive ones, making them well-suited for the complexities often found in programming language syntax. The primary challenges associated with LR parsing often involve the complexity of generating the parse tables and potentially less intuitive error reporting compared to recursive descent.

Evaluation of Rust Parser Generators

Several parser generators are available within the Rust ecosystem, implementing different strategies:

- **Pest:** A popular generator based on PEG parsing. While mature and widely used, its reliance on PEGs deviates from our CFG-based approach for RedType. [27]
- **ANTLR:** A powerful and language-agnostic parser generator framework. While supporting various target languages, its optimal integration and idiomatic usage within the Rust ecosystem have presented challenges in past experiences. [1]
- **nom:** A highly regarded parser combinator library. Instead of defining a grammar in a separate file, 'nom' involves writing Rust functions (combinators) that parse specific parts of the input, which are then combined to parse larger structures. This offers flexibility and performance but lacks a direct, declarative grammar representation, potentially complicating maintenance and direct validation against a formal CFG. Ensuring AST type safety also requires more explicit handling. [4]
- **LALRPOP:** A parser generator specifically designed for Rust that implements the LR(1) strategy (despite being named LALRPOP). It takes a grammar file, performs validation checks for ambiguities and conflicts (based on LR(1) constraints), and generates efficient Rust parsing code. This approach aligns well with traditional compiler construction techniques using CFGs. [18]

Decision: LALRPOP

Based on the evaluation, **LALRPOP** is the most suitable choice for the RedType interpreter project. Our team possesses familiarity with CFGs and bottom-up parsing principles from academic coursework and LALRPOP allows us to directly leverage this knowledge by defining the RedType grammar in a familiar, declarative format.

Key advantages reinforcing this decision include:

- **CFG Alignment:** Directly utilizes an LR(1) grammar specification, fitting our existing grammar work.
- **Grammar Validation:** Incorporates before runtime checks for common LR(1) conflicts, enhancing robustness.
- **Type Safety:** Facilitates the generation of strongly-typed Abstract Syntax Trees (ASTs), simplifying subsequent analysis.
- **Rust Native:** Designed specifically for Rust, ensuring good integration and idiomatic code generation.

While nom offers high performance, its combinator approach diverges from our preference for a declarative grammar. Pest, using PEGs, represents a different parsing paradigm. LALRPOP strikes the optimal balance between parsing power, development methodology alignment, and strong Rust integration for RedType.

LALRPOP Implementation Details

LALRPOP functions by taking a `.lalrpop` grammar file as input. This file defines the terminals, non-terminals, grammar rules, and associated semantic actions. During the build process, LALRPOP analyzes the grammar, constructs the LR(1) parsing tables, and generates a Rust module containing the parser logic. [18], [19]

The generated parser consumes a stream of tokens (produced by a separate lexer, which LALRPOP can also help define) and attempts to derive the start symbol according to the grammar rules. If successful, it returns a value constructed by the semantic actions, often the root of the AST. LALRPOP also provides mechanisms for error reporting and recovery. [18], [19]

By automating the complex process of parser table generation and code implementation based on the proven LR(1) technique, LALRPOP allows developers to focus on the core language design and semantics, making it an invaluable tool for building robust language processors in Rust.

8.3 Summary

In summary, Rust establishes RedType's technical foundation, emphasizing memory safety and execution efficiency. Tokio complements this with capabilities for scalable concurrent processing. LALRPOP contributes to reliable parsing through its declarative, CFG-aligned grammar definition directly within Rust. These choices are intended to support the development of a RedType interpreter that is robust in its type enforcement and efficient in operation.

9 Implementation

This section details the implementation of the RedType language, focusing on key design decisions that enable type-safe database operations, its architectural components, and a comparison of our approach with formal semantics. We emphasize features central to RedType's domain, such as schema-aware type checking and database interaction semantics, while briefly covering more standard language constructs.

9.1 Architecture

The architecture of the RedType language system, illustrated in Figure 3, outlines a sequential processing pipeline for individual requests. The components within the "RedType Server" dotted line operate as internal modules within a single server runtime. This entire Rust server is designed to handle numerous queries concurrently across all stages of this pipeline. The system transforms RedType source code into executable operations, focusing on type safety for database interactions. Figure 3 depicts the standard flow for successful operations. However, an error at any processing stage can result in an immediate error response being returned to the client.

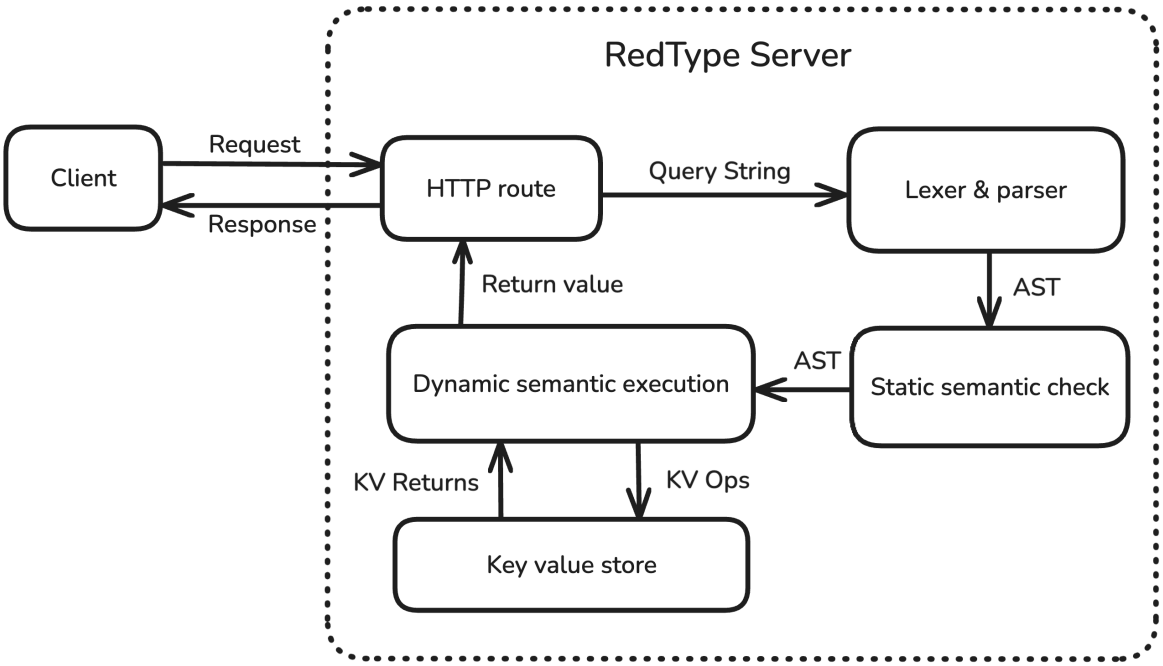

Figure 3: Diagram over RedType's Architecture

When a client request arrives at the HTTP route, its body (containing either a schema update or a query) is passed to the Lexer and Parser. Generated using LALRPOP, this component performs lexical analysis on the input to produce a stream of tokens, which are subsequently parsed according to the language grammar to construct an Abstract Syntax Tree (AST). The AST serves as an intermediate representation, capturing the program's structure.

Following parsing, the AST undergoes Static Semantic Check, primarily focusing on Type Checking. This phase employs a `TypeEnvironment`, which is aware of the database schema, variable types, and function signatures, to verify the program's adherence to RedType's type rules. A key focus here is on schema-aware validation of database operations to catch errors before runtime.

Once the AST is successfully validated, it proceeds to Dynamic Semantic Execution. This component is responsible for executing the program's logic and manages runtime state, including variable values and scopes, within a runtime `Environment`. It interacts with the Key Value (KV) store to perform database operations (indicated by KV Ops and KV Returns). The return value from this execution is then passed back to the HTTP route, which formulates a JSON response to be sent to the client. The Concurrency Strategy, detailed later, explains how Rust's concurrency primitives and Tokio are employed to enable this concurrent processing of requests through all stages.

9.2 Abstract Syntax Tree (AST)

The implementation of the AST is critical for parsing RedType into an unambiguous interpretation of the syntax. The completed AST code is under Appendix F.2. It is implemented with Rust using a series of `struct` and `enum` definitions. These structures closely mirror the productions in the formal grammar (Figure 2), while effectively handling sequences, optionality and recursive data structures. Throughout the implemented AST nodes, there are used derivations for `Debug`, `Clone`, `PartialEq`, `serde::Serialize`, and `serde::Deserialize` for debugging, testing, and potential serialization (for example outputting the AST).

At the AST's highest level, the RedType struct combines optional schema and lock definitions with a sequence of statements as described in the production rule:

$$P ::= \overrightarrow{S}\ \overrightarrow{L}\ \overrightarrow{S_t}$$

```
pub struct RedType {
    pub schemadef: Option<SchemaDefinition>,
    pub lock: Option<Lock>,
    pub statements: Vec<Stmt>,
}
```

Listing 23: Top Level RedType Struct

Here, `Option<T>` denotes optional components (schema or lock might be absent), and `Vec<Stmt>` represents the list of program statements. This pattern of using `Option<T>` for optional grammar elements, and `Vec<T>` for sequences is used throughout the implementation of the AST. For instance schema definitions use a `TypeDefinition`, which contain `Vec<FieldDefinition>` for its fields, and each `FieldDefinition` uses an `Option<Constraint>` for an optional primary key constraint.

Another major consideration for the implementation is recursive data management. This is especially true for the recursive definitions found within `Type` and `Expr` rules. Rust requires that the recursive part is "boxed" using `Box<T>`. This is a more complex type of pointer, that allocates the inner data on the heap and stores a pointer to it within the struct/enum variant [17]. This ensures the outer type has a known, finite size before runtime. For example, the option type uses `Box<CoreType>` to store the nested type on the heap, as `Type` can refer to itself indirectly and thereby cause recursion. Similarly, `Expr` variants that are recursive, like `BinOp { left: Box<Expr>, op: BinOp, right: Box<Expr> }`, use `Box<Expr>` to wrap sub-expressions. This prevents infinite sizing issues before runtime by storing the recursive parts via pointers. Underneath in Listing 24 an example AST of the command: `"SET User[1].name TO "John";"` Can be seen

```
RedType {
    schemadef: None,
    lock: None,
    statements: [
        DatabaseOp(
            Set(
                KeyIdentifier {
                    type_name: "User",
                    key_value: Constant(
                        Int(
                            1,
                        ),
                    ),
                    field_name: Some(
                        "name",
                    ),
                },
                Constant(
                    String(
                        "John",
                    ),
```

```
                ),
            ),
        ),
    ],
}
```

Listing 24: AST Example for SET User[1].name TO "John"

The AST is used by the semantics to interpret and execute RedType programs. This is done by recursively walking through the AST using Rust's `match` expression. Each node type, like `Stmt` or `Expr`, has matching logic that performs the correct action based on the variant. Recursive parts wrapped in `Box<T>` are dereferenced before further processing. This approach allows for a simple and direct way to evaluate the structure of a program.

9.3 Lexing & Parsing RedType With LALRPOP

As covered in 8.2, we use LALRPOP for its ability to define context-free grammars and generate efficient LR(1) parsers. The grammar specifies RedType's syntax, from literals and identifiers to complex statements and expressions, including rules for database operations and schema definitions. This parser transforms RedType source code into an Abstract Syntax Tree, mirroring the language's formal EBNF-like concrete syntax and serving as the foundation for semantic analysis and type checking.

Lexical Analysis

RedType's tokens (terminals) are defined in the LALRPOP grammar using:

- **Regular Expressions:** For variable-form tokens like identifiers (`Ident`) and literals (`IntLiteral`, `StringLiteral`).
- **String Literals:** For fixed keywords (e.g., `"Int"`, `"LOCK"`, `"SET"`) and operators (e.g., `"+"`, `"=="`).

These definitions are then turned into Rust types by actions that convert the matched input text. For example parsing a string into an `i64` type (64-bit integer) for `IntLiteral`, or removing quotes for `StringLiteral`. The default lexer in LALRPOP automatically skips whitespaces, meaning no explicit handling of whitespace is needed in the implementation.

Syntax Analysis and AST Construction

The parsing of RedType is defined by non-terminal rules in the LALRPOP grammar, which are based on the concrete syntax. When these rules are matched, embedded Rust actions are used to construct the corresponding AST node. For example, the top-level `RedType`

rule is defined using LALRPOP's specific syntax, which pairs grammar patterns with Rust code snippets for AST construction:

```
1 pub RedType: RedType = {
2     <schemadef:SchemaDefinition?> <lock:Lock?> <stmts:Stmt*> => RedType {
3         schemadef,
4         lock,
5         statements: stmts
6     }
7 }
```

Listing 25: Top Level RedType Rules

`pub` is a rust keyword to make this RedTypeParser public. This part of the parser returns the `RedType` AST node, which is denoted by the colon on line 1. On line 2 the production rule for $P ::= \overrightarrow{S}\ \overrightarrow{L}\ \overrightarrow{S_t}$ is seen in LALRPOP syntax. Angle brackets (<>) are used to name captured values that stem from a non-terminal or a terminal. For example, `<schemadef:SchemaDefinition?>` returns a named value called `schemadef` based on the result of the non-terminal `SchemaDefinition`, which can be captured zero or one time (marked with:?). Since it is optional, `SchemaDefinition?` becomes `Option<SchemaDefinition>`. The matched values are then used in the action code on the right hand side of `=>`, which in this case creates the RedType struct with all matched values. This is the general method for constructing the AST from each production rule in the concrete syntax. Recursive AST structures, such as expression, utilize the before mentioned Box<T> to allocate the items on the heap.

In addition to optional elements with the `?` operator, LALRPOP has convenient syntax for handling sequences with `*` which wraps the rule with Vec<T> and macros. Custom parameterized macros can be created to capture constructs, for example comma separated lists of items: `Comma<T>`. These constructs allow the LALRPOP grammar to closely match RedType's formal syntax. Remaining LALRPOP grammar can be found at Appendix F.1

Handling Precedence and Ambiguity

LALRPOP requires there to be no ambiguity to be able to parse the production rules and create the AST. This is managed with two primary methods:

- **Structured precedence:** Operator precedence and associativity in expressions (such as the higher precedence of `*` over `+`, and the left-associative behavior of `-`) are defined implicitly by structuring the expression grammar in tiers (e.g., `Expr -> OrExpression -> ... -> Atom`). The concrete syntax has been carefully created to implicitly structure the precedence of rules.

- **Explicit precedence:** For RedType statements, where different individual rules might share common prefixes (e.g, identifier prefixes both declaration and assignment), the `#[precedence(level="...")]` attribute can be applied to explicitly define which rule has precedence based on which level is assigned.

9.4 Static Semantics: Type Checking and Formal Comparison

Static analysis is critical in RedType for pre-runtime error detection, especially for database interactions. Our type checker validates program structure and types against the formal type system.

Type Environment for Schema-Aware Checking

The formal environments Γ_s (schema types), Γ (variable types), and π (functions) are unified in our `TypeEnvironment` struct:

```
pub struct TypeEnvironment {
    // Gamma: Local variable types
    variables: HashMap<String, Type>,
    // Pi: Function signatures
    functions: HashMap<String, FunctionSignature>,
    // Gamma_s: Database schema types
    schema_types: HashMap<String, HashMap<String, Type>>,
    // Context for return type checks
    current_function: Option<FunctionSignature>,
    // For detecting recursive calls
    global_call_stack: Vec<String>,
}
```

Listing 26: TypeEnvironment Struct

The `schema_types` field is crucial, allowing the type checker to validate operations against the defined database schema, a core feature of RedType.

Type Checking Database Operations

To ensure safety, each database operation is governed by a static type-checking rule, ensuring alignment with the declared schema. We formalize these with static semantics rules and demonstrate how they are enforced in the before runtime type checker.

GET Operation

The `GET` operation retrieves a value from the database. Since the key may not exist, the result is wrapped in an `Option<T>` type, requiring the caller to explicitly handle the absence of a value. This design helps avoid null-related bugs as discussed previously in section 6.4.

$$[\text{GET}_{TS}] \quad \frac{\Gamma_s(K) = T}{\Gamma_s,\, \Gamma \vdash x : \texttt{Option<}T\texttt{>} = \texttt{GET}\ \ K : \texttt{ok} \quad \text{and} \quad \Gamma' = \Gamma[x \mapsto \texttt{Option<}T\texttt{>}]}$$

The corresponding implementation in Rust extracts the schema type and annotates the target variable accordingly:

```
check_keyidentifier(source_key, env, false)?;
let field_type = env.get_field_type(&source_key.type_name,
    &source_key.field_name)?;
let option_type = Type::CoreType(CoreType::Option(Box::new(field_type.clone())));
env.add_variable(target_var.clone(), option_type)?;
Ok(())
```

Listing 27: GET Type Check Implementation

This code checks the schema for the key's type, wraps it in an `Option`, and updates the type environment, enforcing that every `GET` result must be pattern-matched before use.

SET Operation

The `SET` operation updates a key in the database. It is only valid if the expression being stored matches the expected type from the schema. This guarantees that runtime type errors cannot occur during updates.

$$[\text{SET}_{TS}] \quad \frac{\Gamma_s(K) = T \quad e : T}{\Gamma_s \vdash \texttt{SET}\ \ K = e : \texttt{ok}}$$

The implementation reflects this rule by checking both the key's schema type and the type of the expression:

```
check_keyidentifier(key, env, false)?;
let expected_type = env.get_field_type(&key.type_name, &key.field_name)?;
let value_type = check_expr(value_expr, env)?;

if !types_compatible(&expected_type, &value_type) { /* Error */ }
Ok(())
```

Listing 28: SET Type Check Implementation

This code ensures the value being assigned aligns with the schema type, thereby maintaining consistency across all database writes.

Pattern Matching on Option Types

To safely handle values returned by `GET`, RedType enforces pattern matching on `Option<T>`. This is done by type checking the match expression and validating each branch.

$$[\text{MATCH}_{TS}] \quad \frac{\begin{array}{l}\Gamma \vdash e : \texttt{Option}\langle T \rangle \\ \forall i \in 1..n,\ \big(\text{if } \mathit{Case}_i = \texttt{Some}(x_i) \Rightarrow S_i,\ \Gamma[x_i \mapsto T] \vdash S_i : \texttt{ok} \\ \quad \wedge\ \text{if } \mathit{Case}_i = \texttt{None} \Rightarrow S_i,\ \Gamma \vdash S_i : \texttt{ok}\big)\end{array}}{\Gamma \vdash \texttt{match}\ e\ \{\texttt{Case}_i\}_{i=1}^{n} : \texttt{ok}}$$

The type checker enforces this rule by ensuring the match expression is an `Option`, and by scoping the bound variable `x` only within the `Some` branch:

```
let expr_type = check_expr(expr, env)?;
if !expr_type.is_option() { /* Error */ }
let inner_type = expr_type.get_option_inner_type()?;

for case in cases {
    match case {
        Case::SomeCase(var_name, stmts) => {
            let mut case_env = env.clone();
            case_env.add_variable(var_name.clone(), inner_type.clone())?;
            check_stmts(stmts, &mut case_env)?;
        },
        Case::NoneCase(stmts) => {
            check_stmts(stmts, env)?;
        },
    }
}
Ok(())
```

Listing 29: Match Type Check Implementation

This approach ensures every match on a `GET` result handles both presence and absence cases explicitly.

Standard Language Constructs

In addition to database-specific operations, RedType includes conventional programming constructs such as variable declarations, assignments, conditionals, and functions. These are type-checked using well-established principles to ensure consistency and soundness throughout the language.

Variable declarations and assignments are validated by checking that the expression on the right-hand side matches the declared or inferred type of the variable. Conditional

statements require the condition to have type `bool`, and both branches must type check under the same environment.

Functions are checked both at their declaration and call sites. Function declarations must specify parameter and return types consistent with the body, and function calls are verified against stored signatures in the `TypeEnvironment`, ensuring that argument types match the expected parameter types.

RedType's static analysis prioritizes catching database-related type errors early. Key aspects include rigorous schema validation for all database operations and ensuring that `Option` types propagate correctly, promoting robust error handling through pattern matching.

9.5 Dynamic Semantics: Interpretation and Formal Comparison

The RedType interpreter evaluates abstract syntax trees by executing statements, managing runtime environments, and interacting with the database. Its behavior aligns closely with the structural operational semantics (SOS) rules defined in section 7.

Runtime Values and Environment

At runtime, RedType represents values using the `Value` enum, which includes primitive types and, critically, the variants `Some(Box<Value>)` and `None` to support optional results from database queries:

```
pub enum Value {
    Int(i64),
    // ...
    Some(Box<Value>),
    None,
}
```

Listing 30: Runtime Value Representation

The runtime environment is captured in the `Environment` struct. This structure maps variable names to runtime `Values` and maintains lexical scope through an optional outer environment. It effectively abstracts the formal store σ and environment $\mathscr{E}$:

```
pub struct Environment {
    variables: HashMap<String, Value>, // Represents sigma and Epsilon
    functions: HashMap<String, Function>, // Represents func environent
    outer: Option<Arc<RwLock<Environment>>>, // Outer scope, used for nested env
    return_value: Option<Value>, // Return value of current environment
}
```

Listing 31: Runtime Environment Structure

Function Definition and Call

As seen in section 7, the following two rules are the dynamic semantics for function definitions and calls.

$$\text{[FUNC-DEF}_{SSS}\text{]} \quad \frac{\pi' = \pi\big[f \mapsto \big((T_1, \ldots, T_n),\, T_{\text{ret}},\, S,\, [x_1, \ldots, x_n],\, \mathscr{E},\, \pi\big)\big]}{\mathscr{E}, \pi \;\vdash_l\; \big\langle\; \texttt{func}\;\; f(x_1 : T_1, \ldots, x_n : T_n) : T_{\text{ret}}\{S\},\; \sigma \big\rangle \;\rightarrow\; \langle \sigma, \pi' \rangle}$$

$$\text{[FUNC-CALL}_{SSS}\text{]} \quad \frac{\begin{array}{l} \pi(f) = \big(S,\, [x_1, \ldots, x_n],\, \mathscr{E}',\, \pi'\big), \\ \forall i \in \{1..n\}.\, \mathscr{E}(y_i) = \ell_i,\; \sigma(\ell_i) = v_i, \\ \text{fresh locations } \ell_i' \text{ for each } i \text{ where } v_i \text{ is not an array}, \\ \mathscr{E}'' = \mathscr{E}'\left[x_i \mapsto \begin{cases} \ell_i & \text{if } v_i \text{ is an array} \\ \ell_i' & \text{otherwise} \end{cases} \;\middle|\; i \in \{1..n\}\right], \\ \sigma' = \sigma\left[\ell_i' \mapsto v_i \;\middle|\; v_i \text{ is not an array}\right], \\ \pi'' = \pi'[f \mapsto (S, [x_1, \ldots, x_n], \mathscr{E}', \pi')], \\ \mathscr{E}'', \pi'' \;\vdash_l\; \langle S, \sigma' \rangle \rightarrow \sigma'' \end{array}}{\mathscr{E}, \pi \;\vdash_l\; \langle f(y_1, \ldots, y_n), \sigma \rangle \rightarrow \sigma''}\, t$$

To handle function definitions the function below in listing 32 has been defined for collecting them before execution, and storing them in the `Environment` struct previosly shown in listing 31.

```
fn collect_function_definitions(
    &self,
    stmts: &[Stmt],
    env: Arc<RwLock<Environment>>,
) -> Result<(), RuntimeError> {
    for stmt in stmts {
        if let Stmt::FunctionDef(func_def) = stmt {
            let function = Function {
                name: func_def.name.clone(),
                params: func_def.params.clone(),
                body: func_def.body.clone(),
                return_type: func_def.return_type.clone(),
            };
            env.write().unwrap().define_function(function);
        }
    }
    Ok(())
}
```

Listing 32: Runtime Environment Structure

This function in Listing 32 iterates over all statements in a program and for each function definition, it creates a `Function` struct with the associated `name`, `params`, `body`, and `return_type`. The function is thereafter saved in the environment on line 14 using `define_function`. After collecting all functions, the interpretation will continue with evaluating the remaining statements in a program. Below in listing 33, is the case for when an expression evaluates to a function call:

```
1  Expr::Call(func_name, args) => {
2      if self.is_builtin_function(func_name) {
3          return self.call_builtin_function(func_name, args, Arc::clone(&env));
4      }
5
6      let func = env.read().unwrap().get_function(func_name)?;
7
8      let mut arg_values = Vec::new();
9      for arg in args {
10         let value = self.evaluate_expr(arg, Arc::clone(&env))?;
11         arg_values.push(value);
12     }
13
14     // Check arity (parameter count validation), cut off to shorten
15
16     let func_env_arc = Arc::new(RwLock::new(Environment::with_outer(Arc::clone(&env))));
17
18     for (param, arg_val) in func.params.iter().zip(arg_values) {
19         func_env_arc.write().unwrap().define_var(param.name.clone(), arg_val);
20     }
21
22     let result = self.execute_stmts(&func.body, Arc::clone(&func_env_arc))?;
23
24     if func_env_arc.read().unwrap().has_returned() {
25         Ok(func_env_arc.read().unwrap().get_return_value().unwrap())
26     } else {
27         match result {
28             Some(val) => Ok(val),
29             None => Ok(Value::None),
30         }
31     }
32 }
```

Listing 33: Runtime Environment Structure

This specific match case `Expr::Call` is part of a larger function called `evaluate_expr`. This `evaluate_expr` function contains a large match case featuring each of the different

expression types, where the case for calling a function can be seen in the code snippet above. When this match case is hit, it first checks if the called function is a built-in function (lines 2-4 in the snippet); if so, `selfċall_builtin_function` is invoked and its result returned. If the function is not built-in, a read lock on the environment allows fetching the function definition (line 6). Subsequently (lines 8-12), arguments are evaluated by `selfėvaluate_expr` into Value objects. These Values are collected, and their internal structure handles pass-by-reference (e.g., for arrays, via shared ownership through Arc) or pass-by-value semantics. A comment on line 14 notes an arity check placeholder. With argument values captured, a new lexically scoped function environment is created (line 16). Then, per the [FUNC-CALL] rule, these evaluated arguments are bound to the function's formal parameters `funcṗarams` within this new environment (lines 18-20). The function body is executed via `selfėxecute_stmts` (line 22). Finally (lines 24-31), if the function's environment indicates an explicit return occurred, that value is returned; otherwise, the result of the body's execution (if it yielded a value) or Value::None is returned.

Executing Database Operations and Core Constructs: GET Operation

The SOS rule for reading from the database is:

$$[\text{GET}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB(K) = v}{\langle x = \texttt{GET}\ K,\ DB,\ \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma[x \mapsto \texttt{Some}(v)]\rangle}$$

If the key is not found in the database, the variable is instead bound to `None`. This dynamic treatment of optionality complements the static typing rules.

The corresponding interpreter logic retrieves a value from the in-memory database and binds it to the target variable as either `Some(v)` or `None`:

```
1  let key_str = evaluate_and_form_db_key(source_key, env)?;
2  let db_guard = self.db_store.read().unwrap();
3  let retrieved_value = db_guard
4      .get(&source_key.type_name) // and so on
5      .map_or(Value::None, |db_val| {
6          Value::Some(Box::new(convert_from_db_value(db_val)))
7      });
8
9  env.borrow_mut().define_var(target_var.clone(), retrieved_value);
10 Ok(None)
```

Listing 34: GET Operation Execution

SET Operation

The SOS rule for setting a database value is:

$$[\text{SET}_{SSS}] \quad \frac{\sigma \vdash e \rightarrow_{\text{Aexp}} v \quad DB' = DB[K \mapsto v]}{\langle \texttt{SET } K \texttt{ TO } e,\ DB,\ \sigma \rangle \Rightarrow \langle \texttt{skip},\ DB',\ \sigma \rangle}$$

The interpreter evaluates the expression, converts the result to a storable database format, and inserts it into the internal database representation:

```
let key_str = evaluate_and_form_db_key(key, env)?;
let value_to_set = self.evaluate_expr(value_expr, env)?;
let db_value = convert_to_database_value(&value_to_set)?;

let mut db_guard = self.db_store.write().unwrap();
// Insert db_value into nested database structure
Ok(None)
```

Listing 35: SET Operation Execution

This behavior directly models the store update in the formal semantics, ensuring the interpreter's state changes reflect the theoretical rules defined in the SOS system.

9.6 Concurrency Strategy

Concurrency in RedType is introduced at the server level. By parallelizing CPU-bound tasks, the server can efficiently handle multiple incoming queries, performing syntax validation and AST generation concurrently. This is especially important for maintaining responsiveness under heavy load, particularly when processing complex queries involving database interactions. As previously discussed in section 8.1, concurrency is primarily managed using the Tokio runtime.

The concurrency strategy relies on a single, shared in-memory database store, referred to as DatabaseStore, which is implemented as a HashMap. All threads access this shared data structure safely through a combination of two Rust primitives: Arc and RwLock. [17]

An Arc, or atomically reference-counted pointer, allows multiple threads to hold shared ownership of the DatabaseStore without duplicating the data. Each clone of the Arc increments a thread-safe reference count, ensuring that the data remains alive as long as any thread still holds a reference. Once the last reference is dropped, the memory is safely deallocated. However, since the database is consistently held by the server, it remains allocated for that duration. [17]

The DatabaseStore itself is wrapped in an RwLock, a readers–writer lock that allows multiple threads to acquire read access simultaneously, ideal for scenarios where data is fre-

quently read but infrequently written. When a thread needs to modify the data, it must acquire a write lock, which excludes all other readers and writers for the duration of the update. This implementation comes with a downside since only one thread can write at a time. This could be resolved by following an approach similar to DragonflyDB, namely being a sharding of the database mentioned in Section 2.6, so multiple `RwLock` would be available. [17]

Together, this pattern `Arc<RwLock<DatabaseStore>>` provides a solution for a shared mutable state, enabling safe concurrent reads and synchronized, exclusive writes across threads.

```
// Database store type
pub type DatabaseStore = HashMap<String, HashMap<String, HashMap<String,
    DatabaseValue>>>;
// Wrapped in Arc<RwLock<>>> to make it thread safe
Arc<RwLock<DatabaseStore>>
```

Listing 36: Thread safe DatabaseStore example

The same pattern is used for key locks to make the locking process thread-safe. The implementation follows the pessimistic approach, as covered in Section 6.9, and focuses on locking individual keys rather than the whole system. The locked keys are stored in the set `key_locks`, which is safely shared using Arc and RwLock. When a new thread wants to access an already locked key, it is added to a queue of pending requests called `pending_locks` and then notified by `lock_notify` when the key is unlocked. This can be seen in the following listing:

```
pub struct Interpreter {
    // ...
    db_store: Arc<RwLock<DatabaseStore>>,
    // Key-specific locks instead of a global lock
    key_locks: Arc<RwLock<HashMap<KeyLock, Arc<TokioMutex<()>>>>>,
    // Queue of pending lock requests
    pending_locks: Arc<RwLock<VecDeque<PendingLock>>>,
    // Notify mechanism for lock availability
    lock_notify: Arc<Notify>,
}
```

Listing 37: Shared key locking architecture

Each key lock within the key_locks map is an `Arc<TokioMutex<()>>`. A `TokioMutex` is similar to a normal mutex, but it is also asynchronous and non-blocking. This means that if the lock is not available, it allows the current task to yield, freeing the thread to execute

other tasks. The `Arc` around the `TokioMutex` allows multiple tasks that need to operate on the same specific key to await its lock.

9.7 Summary

The implementation of RedType unites several general and domain-specific techniques to support schema-aware, type-safe interactions with an in-memory database. By structuring the system as a processing pipeline, spanning lexing, parsing, static analysis, and interpretation, the language is able to enforce correctness properties before execution. The use of LALRPOP for parser generation and Rust's type system for encoding recursive syntax trees, demonstrates how these tools are effective for the language construction of RedType.

While RedType's implementation features are still early in their development and could benefit from further refinement, this lays a promising foundation for exploring the integration of schema and programmatic constructs within a unified language. Overall, the current architecture serves as a functional basis for RedType, but further iterations are needed to improve modularity and robustness before RedType can be considered mature or broadly applicable.

10 Testing

Testing was a critical component of our development process, ensuring the reliability and correctness of the Redtype implementation. We developed a testing strategy that systematically addressed each component of the RedType architecture, as detailed in Section 9.1 (see Figure 3 in particular). Our approach covered multiple levels of the system, from the foundational components like the Lexer and Parser (responsible for initial code processing), through the Static Analysis (including Type Checking) and Interpreter (handling dynamic semantics and database interactions with the `DatabaseStore`), to complete system-level exploratory testing. This multi-faceted strategy aimed to verify not only individual component correctness but also their integration and the overall system behavior. Our approach combined traditional unit testing with integration and system-level validation to create a verification framework. This section outlines our testing methodology for each component of the language implementation, detailing how these efforts contributed to the overall quality of the final product, including efforts to test all token types, selected critical value ranges, and some production rules from our LALRPOP grammar.

10.1 Testing Methodology

Our test framework, built in Rust, utilized custom assertion macros to compare the Abstract Syntax Tree (AST) produced by the Parser against expected structures. This provided detailed error messages when discrepancies were found, allowing us to verify not only that parsing succeeded but also that the resulting AST correctly represented the semantic structure of the input code. These tests were important in driving the development of the Lexer and Parser stages of our architecture, helping us refine the grammar and ensuring that syntax changes did not introduce regressions. We created tests covering standard use cases, edge conditions (e.g., empty inputs, boundary values), and error conditions (e.g., malformed syntax). In total, our comprehensive test suite includes 120 test functions across 9 test modules, with fine-grained unit tests for lexer and parser rules, and broader integration tests for semantic analysis and system operations. For instance, potentially problematic areas like operator precedence, recursive type definitions, and complex database operation scenarios received particular attention with dedicated test cases.

10.2 Unit & Integration testing

This section outlines our testing strategy for unit tests and integration tests to ensure effective collaboration between individual components. We tested key areas such as the lexer, parser, static semantic analysis (type checking), and dynamic semantic evaluation (runtime behavior) to guarantee the language's reliability and adherence to its design.

Lexer and Parser

Our testing strategy for the lexer and parser components employed a suite of unit tests that verify the correct tokenization and parsing of language constructs. These tests were organized into distinct modules, each focusing on a specific aspect of the language syntax.

For expression testing, we validated the parser's ability to correctly interpret various expression types ranging from simple literals (integers, doubles, booleans, strings) to complex constructs like arithmetic operations with proper operator precedence, logical operations, function calls, array literals and access, and unary operations. Listing 38 below provides an example of such a test. This test, marked by the `#[test]` attribute (line 1), focuses on arithmetic expression parsing with correct operator precedence. It defines the expected AST structure for an expression like `(3 + 5) * 2` (lines 3-11). Within this structure, the `Expr::BinOp` representing the multiplication (line 3) contains another nested `Expr::BinOp` for the addition `(3 + 5)` as its left child (lines 4-8). This ensures that the addition is evaluated before the multiplication, correctly reflecting operator precedence. The `assert_exp_eq!` macro (line 12) then compares the AST generated by parsing the string `(3 + 5) * 2` against this predefined complex AST structure, ensuring the parser correctly captures the intended semantic hierarchy. Similarly, statement tests ensured correct parsing of language statements including variable declarations and assignments, control flow structures such as if-else statements, for loops, while loops, and database operations.

```
1  #[test]
2  fn complex_arithmetic_with_precedence1() {
3      let complex = Expr::BinOp {
4          left: Box::new(Expr::BinOp {
5              left: Box::new(Expr::Constant(Constant::Int(3))),
6              op: BinOp::Add,
7              right: Box::new(Expr::Constant(Constant::Int(5))),
8          }),
9          op: BinOp::Mul,
10         right: Box::new(Expr::Constant(Constant::Int(2))),
11     };
12     assert_exp_eq!("(3 + 5) * 2", complex);
13 }
```

Listing 38: Example test for parsing arithmetic expressions with precedence

We also implemented dedicated tests for type parsing, verifying that both primitive types (Int, Double, String, Bool) and collection types (arrays) were correctly recognized. Database operation testing confirmed proper parsing of specialized operations including SET, DEL, INCR, DECR, and GET commands with their respective syntax requirements. As shown in Listing 39, we verified that database SET operations were correctly parsed into their corresponding AST representation. This test (marked `#[test]` on line 1) defines the expected AST for the command `SET User[1].message TO "Hello World";` (lines 3-11). The

`DatabaseOp::Set` enum variant (line 3) encapsulates a `KeyIdentifier` (lines 4-8) and the value being set. The `KeyIdentifier` specifies the entity type (`User`), key (`1`), and field (`message`). The value is a string constant `"Hello World"` (line 9), created using a helper `str_const`. The `assert_dataop_eq!` macro (line 11) then ensures that parsing the command string correctly produces this AST structure, confirming that operations like `SET User[1].message TO "Hello World";` were properly handled by the Parser component of our architecture.

```
1  #[test]
2  fn test_set_operation() {
3      let expr = DatabaseOp::Set(
4          KeyIdentifier {
5              type_name: "User".to_string(),
6              key_value: int_const(1),
7              field_name: Some("message".to_string()),
8          },
9          str_const("Hello World"),
10     );
11     assert_dataop_eq!("SET User[1].message TO \"Hello World\";", expr);
12 }
```

Listing 39: Example test for parsing a database SET operation

The testing approach combined both positive tests validating correct syntax with negative tests ensuring appropriate error handling for invalid input. For each language construct, we tested basic valid syntax with simple expressions, complex nested structures, edge cases like empty arrays and boundary values, and error cases to verify the parser correctly rejected malformed input.

Static Semantics and Dynamic Semantics

For semantic components, our testing was divided into two main categories: static semantic analysis also known as type checking and dynamic semantic evaluation which is the runtime behavior.

Static Semantic Testing

Our static semantic tests focused on validating the type system and ensuring proper detection of semantic errors before program execution. Type checking verification was a central aspect of this, confirming that operations are performed only on compatible types and that type mismatches, such as adding an integer to a string or using a non-boolean value in conditional statements, are properly detected and reported. Listing 40 illustrates our approach to testing type mismatches. The `test_type_mismatch` function (line 2), marked as a test with `#[test]` (line 1), constructs a RedType program snippet designed to fail type checking (lines 3-17). It first declares an integer variable ‘x‘

initialized to ‘5‘ (lines 4-7). Then, it attempts to assign the result of ‘$x + "hello"$‘ to a variable ‘y‘ (lines 9-17). This operation, adding an `Int` to a `String`, is a type error. The `create_simple_redtype` utility (line 3) assembles these statements. The core of the test involves calling `check_redtype(&redtype)` (line 19), which is the entry point to our Static Analysis (Type Checking) component. We assert that this check returns an error (`result.is_err()`, line 20), indicating the type mismatch was detected. Finally, the test verifies that the error is indeed a `TypeErrorKind::TypeMismatch` (lines 22-26), confirming the correct error type is reported by our system, demonstrating how our system correctly identifies and reports an attempt to add a string to an integer.

```
1  #[test]
2  fn test_type_mismatch() {
3      let redtype = create_simple_redtype(vec![
4          Stmt::Declare(
5              "x".to_string(),
6              Type::CoreType(CoreType::Primitive(PrimitiveType::Int)),
7              Expr::Constant(Constant::Int(5)),
8          ),
9          Stmt::Assign(
10             "y".to_string(),
11             Expr::BinOp {
12                 left: Box::new(Expr::Identifier("x".to_string())),
13                 op: BinOp::Add,
14                 right: Box::new(Expr::Constant(Constant::String("hello".to_string()))),
15             },
16         ),
17     ]);
18
19     let result = check_redtype(&redtype);
20     assert!(result.is_err(), "Expected an error for type mismatch");
21
22     if let Err(errors) = result {
23         let found = errors
24             .iter()
25             .any(|e| matches!(e.kind, TypeErrorKind::TypeMismatch { .. }));
26         assert!(found, "Expected TypeMismatch error not found");
27     }
28 }
```

Listing 40: Example test for static type checking that detects a type mismatch

The tests also verified proper enforcement of variable scoping rules, including detection of undefined variables and duplicate declarations. Function validation was thoroughly tested to ensure that function calls match their definitions in terms of parameter types and counts, and that return types are consistent with function declarations.

Schema type validation testing confirmed that custom defined schema types are properly validated when used in the program, while array operation tests verified type checking for array declarations, access, and operations, ensuring array indices and element types are correct. A particularly important area of testing was recursive function analysis, which verified the detection of circular references in function calls, distinguishing between valid self-recursion and problematic mutual recursion patterns.

The static semantic tests were designed to ensure that all type errors are caught during the analysis phase, providing meaningful error messages that help developers identify and fix issues in their code before runtime.

Dynamic Semantic Testing

The dynamic semantic tests evaluated the runtime behavior of correctly-typed programs, verifying that expressions evaluate as expected and statements execute properly. Expression evaluation testing verified that arithmetic, logical, and comparison expressions produce correct results according to the language specification.

Variable management tests confirmed proper declaration, assignment, and retrieval of variables from the runtime environment. Control flow testing validated that conditional statements and loops correctly control program execution based on runtime values.

Function execution testing confirmed that function calls properly pass arguments, execute the function body, and return results as expected. Recursive function calls were specifically tested to ensure they work correctly, using examples like factorial calculation to verify proper handling of the call stack. Listing 41 shows our test for a recursive factorial implementation. The `test_function_recursive` function (line 2), marked `#[test]` (line 1), begins by initializing a `RedTypeParser` (line 3), part of our Lexer and Parser architecture. It then defines a RedType program as a raw string (lines 5-15). This program includes a recursive `factorial` function (lines 6-12) and a call to this function, `factorial(5)`, with the result stored in a variable `result` (line 14). The `parser.parse(program).unwrap()` call (line 17) converts this string into an AST. An `interpreter` is created using a test utility (line 18), and its `interpret(&ast)` method (line 19) executes the program. This engages the Interpreter component, which handles dynamic semantics. Finally, the test retrieves the value of `result` from the interpreter's environment (line 21) and asserts that it equals `Value::Int(120)` (line 22), verifying that the function correctly calculates the factorial of 5 by properly managing the recursive calls.

```
#[test]
fn test_function_recursive() {
    let parser = RedTypeParser::new();

    let program = r#"
        func factorial(n: Int): Int {
            if (n <= 1) {
                return 1;
            } else {
                return n * factorial(n - 1);
            }
        }

        result: Int = factorial(5);
    "#;

    let ast = parser.parse(program).unwrap();
    let interpreter = create_interpreter();
    let _result = interpreter.interpret(&ast).unwrap();

    let result_val = interpreter.get_variable("result").unwrap();
    assert_eq!(result_val, Value::Int(120));
}
```

Listing 41: Example test for recursive function evaluation at runtime

Database operations testing verified that interactions with the underlying storage system worked correctly, ensuring commands like SET, GET, and DEL performed as expected. Array handling tests confirmed proper creation, element access, and iteration over arrays using our built-in array functions. Listing 42 demonstrates how we tested array operations at runtime. The `test_array_operations` function (line 2), marked `#[test]` (line 1), initializes a `RedTypeParser` (line 3). It then defines a RedType program (lines 5-10) that declares an integer array `arr` (line 6), accesses its first element using the `get` function (line 7), modifies its third element using the `insert` function (line 8), and accesses the modified element (line 9). The program is parsed into an AST (line 12), and an `interpreter` is created (line 13). The `interpreter.interpret(&ast)` call (line 14) executes these array operations by interacting with the Interpreter and its runtime environment. The test then retrieves the values of `first` (line 16) and `third` (line 17) from this environment and asserts their correctness (`Some(10)` and `Some(35)` respectively, lines 19-20), verifying that array elements could be accessed and modified correctly through our type-safe array interface.

```
#[test]
fn test_array_operations() {
    let parser = RedTypeParser::new();

    let program = r#"
        arr: Int[] = [10, 20, 30, 40, 50];
        first: Option<Int> = get(arr, 0);
        insert(arr, 2, 35);
        third: Option<Int> = get(arr, 2);
    "#;

    let ast = parser.parse(program).unwrap();
    let interpreter = create_interpreter();
    let _result = interpreter.interpret(&ast).unwrap();

    let first_val = interpreter.get_variable("first").unwrap();
    let third_val = interpreter.get_variable("third").unwrap();

    assert_eq!(first_val, Value::Some(Box::new(Value::Int(10))));
    assert_eq!(third_val, Value::Some(Box::new(Value::Int(35))));
}
```

Listing 42: Example test for array operations at runtime

Additionally, error handling tests ensured proper runtime error detection and reporting for issues that cannot be caught during static analysis.

Our testing approach for both static and dynamic semantics involved a combination of simple unit tests for individual features and integration tests that verified the interaction between different language components. This comprehensive testing strategy was crucial for ensuring that the language semantics were robust and consistent, providing a reliable foundation for developers using our language.

10.3 System Test

Our system tests focused on validating end-to-end functionality through server-based integration tests designed to verify that the entire system operates correctly when all components interact.

The primary system test suite implements a series of tests that exercise the server's API endpoints using the Axum testing framework to simulate HTTP requests and verify the responses. We tested basic server health checks to ensure the service is running properly, schema application and validation to confirm that schemas are correctly parsed and applied to the database, and data manipulation through the command API to verify that database operations function correctly. Listing 43 shows a representative system test. This

asynchronous Tokio test (`#[tokio::test]` on line 1), `test_schema_preservation` (line 2), first sets up a test server instance (line 3), which encompasses the entire RedType system architecture. A simple schema for a `User` entity with an `id` and `name` is defined (lines 5-10). An HTTP POST request is made to the `/schema` endpoint to apply this schema (lines 12-17), testing the server's schema processing. The test asserts an `OK` status code (line 18). Next, a RedType command to `SET User[1].name TO "John"` is defined (lines 21-23) and sent via POST to the `/command` endpoint (lines 24-28). This validates the server's command execution logic, which involves the Interpreter and its `DatabaseStore`. Again, an `OK` status is asserted (line 30). To verify data persistence, a GET request is made to `/dbStats` (line 32), and its JSON response is parsed (line 35). The test concludes by asserting that the `User` entity exists in the database statistics and contains one entry (line 37), confirming the successful application of the schema and persistence of data through the integrated system.

```
1  #[tokio::test]
2  async fn test_schema_preservation() {
3      let server = setup_test_server().await;
4
5      let schema = r#"
6      User {
7          id: Int @primary,
8          name: String
9      }
10     "#;
11
12     let response = server
13         .post("/schema")
14         .add_header("Content-Type", "text/plain")
15         .text(schema)
16         .await;
17
18     response.assert_status(StatusCode::OK);
19
20     let command = r#"
21     SET User[1].name TO "John";
22     "#;
23
24     let response = server
25         .post("/command")
26         .add_header("Content-Type", "text/plain")
27         .text(command)
28         .await;
29
30     response.assert_status(StatusCode::OK);
31
32     let response = server.get("/dbStats").await;
```

```
33      response.assert_status(StatusCode::OK);
34
35      let json_data: Value = response.json();
36
37      assert_eq!(json_data["entities"]["User"], 1);
38  }
```

Listing 43: Example system test for schema application and data persistence

Schema preservation logic testing confirmed that data persists when a compatible schema is reapplied, while schema evolution behavior tests validated that the system handles schema changes appropriately. One particularly important test case verified the system's behavior when a schema changes, confirming that when a schema evolves (for example, by adding a new field to an entity), the system correctly resets affected data structures. This ensures that the database maintains consistency between its schema definition and the stored data.

Our system tests provide confidence that the server correctly implements the interface between the language components (lexer, parser, semantic analyzers) and the database system. These tests were instrumental in catching integration issues that were not apparent during unit testing of individual components, ensuring a robust and reliable system.

Exploratory Test

Our exploratory testing was a crucial component of our validation process, providing insights that went beyond structured test cases by examining the system in a more free-form manner. This approach was particularly valuable for validating how TypeScript clients integrate with RedType.

TypeScript Client Testing

For exploratory testing of TypeScript clients, we developed a comprehensive suite of experiments to understand how a client interacts with the RedType server in real-world scenarios. The client implementation was designed to provide a developer-friendly interface to RedType's functionality while maintaining the type safety guarantees that are central to the language.

The TypeScript client, as shown in Listing 44, allows frontend applications to communicate with the RedType server through an intuitive API. The example defines an asynchronous function `reserveProduct` (line 5) that takes a `productId` and `quantity`. It instantiates a `RedTypeClient` pointing to the server (line 3). A RedType script is constructed as a template literal (lines 6-13), demonstrating how locking for `stockAvailable` and `stockReserved` (lines 7-8) would be initiated before product reservation logic. The

`client.executeScript` method (line 16) sends this script to the RedType server. The function includes basic error handling for the script execution (lines 18-20). This setup was used to manually test various scenarios, such as concurrent reservation attempts, network interruptions, and different script complexities to observe the client-server interaction and identify potential issues not covered by automated tests.

```
1  import { RedTypeClient } from "redtype-client";
2
3  const client = new RedTypeClient("http://redtype.server:1337");
4
5  async function reserveProduct(productId: string, quantity: number): Promise<string> {
6    const scriptContent = `
7      LOCK Product["${productId}"].stockAvailable,
8           Product["${productId}"].stockReserved;
9
10     // Reservation logic here...
11
12     return "SUCCESS: Items reserved.";
13   `;
14
15   try {
16     const result = await client.executeScript(scriptContent);
17     return result as string;
18   } catch (error) {
19     console.error("Error executing RedType script:", error);
20     return "ERROR: Reservation failed.";
21   }
22 }
```

Listing 44: Example of exploratory test for product reservation in Typescript using Redtype

Our exploratory testing revealed several insights:

1. **Error Handling**: We discovered edge cases in how network errors and server-side type errors were communicated back to the client. These findings led to improved error handling in the client library.

2. **Connection Management**: Testing under various network conditions helped refine connection retry logic and timeout handling.

3. **Script Generation**: We experimented with different approaches to constructing RedType scripts from TypeScript, including template literals and a builder API, leading to a more robust client interface.

4. **Performance Characteristics**: Manual testing with variably sized scripts and different operation types revealed performance patterns that informed optimization efforts.

This free-form investigation complemented our formal testing by uncovering real-world usage patterns and potential issues that might not have been evident from structured tests alone.

10.4 Evaluation and Performance Tests

Acceptance testing was conducted to validate that RedType satisfies the requirements defined in our MoSCoW prioritization framework. This testing phase focused on ensuring that the implemented features align with the requirements.

MoSCoW Requirement Validation

Acceptance testing was conducted to validate that RedType satisfies the requirements defined in our MoSCoW prioritization framework. For this section, no external stakeholder was available to conduct formal acceptance testing. Instead, the evaluation was performed internally by the development team based on the predefined MoSCoW requirements. The testing phase focused on ensuring that the implemented features align with user expectations and requirements, based on the final state of development.

For each "Must Have" requirement, we designed specific acceptance tests:

1. **Schema Definition Language (Req. 1)**: We validated that the system correctly supports all primitive types (String, Int, Double/Float, Boolean) through a series of schema creation and validation tests.
2. **Server-side Schema Enforcement (Req. 2)**: Tests confirmed that data operations respect schema constraints, with appropriate rejection of non-conforming data.
3. **Type-safe Query Language (Req. 3)**: We verified the language's ability to detect type errors prior to execution through static analysis, using both valid and invalid query scenarios.
4. **Key-value Data Store with Type Enforcement (Req. 4)**: Tests confirmed proper storage and retrieval of all primitive data types with consistent type checking.
5. **CRUD Operations Support (Req. 5)**: We validated that the query language properly supports Create, Read, Update, and Delete operations across all data types.
6. **Filtering Support (Req. 6)**: Tests verified the ability to perform filtering operations with multiple conditions using comparison and logical operators.
7. **Field Selection Support (Req. 7)**: We confirmed the language's support for limiting returned data through field selection.

8. **Atomic Query Execution (Req. 8)**: Tests validated that queries execute atomically when using locks, preventing race conditions and ensuring data consistency.

For "Should Have" requirements, validation was as follows:

9. **Collection Types Support (Req. 9)**: Support for List and Set collection types was deferred to maintain focus on core language design and key-value pair operations within the project's scope. Therefore, tests for this feature were not applicable.

10. **Periodic Snapshots (Req. 10)**: Periodic snapshots were not implemented as the primary focus remained on language design, and this feature fell out of scope. Consequently, no acceptance tests were performed for this requirement.

11. **Concurrent Atomic Queries (Req. 11)**: We validated that the system supports concurrent operations through key-level locking instead of global locks.

For the "Could Have" requirements, the status is:

12. **Alternative Response Encodings (Req. 12)**: Alternative response encodings were not implemented due to a focus on other core features. As such, acceptance tests were not conducted for this item.

13. **Binary Primitive Type (Req. 13)**: The binary primitive type was not implemented as development priorities focused on other areas of the language. Therefore, this feature was not subjected to acceptance testing.

14. **Multi-threading Support (Req. 14)**: Multi-threading support for handling concurrent client connections is inherently provided by the Tokio runtime used in the server implementation. While the system benefits from this by design, specific acceptance tests to benchmark or validate this aspect beyond general server responsiveness were not prioritized for this phase.

The acceptance testing results confirmed that RedType successfully implements all "Must Have" requirements. Additionally, the "Should Have" requirement for Concurrent Atomic Queries was validated, and Multi-threading Support, a "Could Have" requirement, is inherently part of the system's design. Other "Should Have" and "Could Have" requirements were deferred or not implemented to maintain focus on core language design. This provides a solid foundation for a typesafe Redis-like database system. The implemented features demonstrate the system's ability to prevent type-related errors while maintaining performance characteristics necessary for real-time data operations.

Benchmarking Redtype versus Redis

As part of our acceptance testing, we conducted comparative performance benchmarks between RedType and Redis with Lua. It is important to note that Redis is a highly

optimized, production-grade database system with thousands of contributed development hours from a large community of developers. In contrast, RedType is a newer academic project primarily focused on type safety rather than raw performance optimization.

The performance testing was conducted as an exploratory exercise to understand the current performance characteristics of RedType compared to an established industry standard. The methodology involved initializing both Redis (connecting to `redis://127.0.0.1:6379`) and RedType (connecting to `http://127.0.0.1:1337`) instances. For RedType, a specific schema was applied:

```
User {
    id: Int @primary,
    name: String,
    age: Int
}
```

This schema defined a `User` entity with an integer primary key `id`, a string field `name`, and an integer field `age`.

The benchmarks covered two main categories of operations:

- **Single Operations**: Individual SET, GET, and DEL operations were tested. For Redis, these were executed using Lua scripts. For RedType, equivalent commands were sent to the `/command` endpoint. For example, a RedType SET operation involved locking the specific field and then setting its value (e.g., `LOCK User[1].name; SET User[1].name TO "User 1";`).
- **Computational Query**: A more complex query was executed over 10,000 records. For Redis, a Lua script calculated the average, minimum, and maximum age, and a count of users. For RedType, a script with a `while` loop iterated through user IDs, performed GET operations for the age, and computed the same aggregate statistics.

All test results are an average of 100 runs and done on a Macbook M1 Max. Importantly, the RedType benchmarks utilize concurrent request batching where multiple operations are sent simultaneously using `tokio::spawn` tasks, with configurable batch sizes (default 1,000 concurrent requests per batch), simulating real-world multi-client scenarios. These benchmarks provide valuable insights into areas where future optimization might be beneficial, while also highlighting scenarios where RedType's design decisions may offer performance advantages.

Single Operations (10 000 writes/reads/deletes)

Operation	Redis	RedType	Ratio (RedType / Redis)
SET	0.111 s	0.212 s	1.91 ×
GET	0.107 s	0.230 s	2.15 ×
DEL	0.105 s	0.229 s	2.19 ×

Table 3: Performance of individual SET, GET and DEL commands

As shown in Table 3, Redis with Lua outperforms RedType in individual operation speed. RedType is observed to be slower across all basic operations: approximately 1.91 times slower for SET, 2.15 times slower for GET, and 2.19 times slower for DEL operations. Notably, the performance overhead in RedType is evident across all operations, including GETs which are 2.15 times slower. This performance gap across all operations is expected and can be attributed to several factors. Redis benefits from a highly optimized C implementation, designed with raw performance in mind, while RedType prioritizes type safety, which naturally introduces some performance trade-offs. Additionally, RedType incurs extra overhead due to its type checking and validation mechanisms that are executed with each operation. Furthermore, Redis has a mature and finely tuned memory management system along with advanced data structure optimizations that contribute to its superior speed.

Computational Query (10,000 records)

Metric	Redis	RedType	Ratio (RedType / Redis)
Latency	0.009 s	0.013 s	1.45 ×

Table 4: Complex data processing performance

In Table 4, we see that Redis now outperforms RedType on complex computational queries over 10,000 records. Redis completes the query in just 0.009 seconds, whereas RedType requires 0.013 seconds, giving a latency ratio (RedType / Redis) of 1.45×. In other words, RedType is approximately 45 % slower than Redis for this workload.

These benchmarks demonstrate that while RedType doesn't match Redis's raw performance for simple operations, its architecture shows promise for complex analytical workloads where its type safety features provide both correctness guarantees and potential performance benefits. The results validate that RedType's primary goal of providing type safety doesn't necessarily come with prohibitive performance costs in all scenarios.

It is worth noting that these benchmarks represent a point-in-time comparison rather than a definitive performance assessment. As RedType matures, targeted optimizations could potentially narrow the performance gap for basic operations while maintaining or expanding its advantage in computational scenarios.

Performance metrics of Redtype

The performance metrics against Redis showcases a comparison, but does not pinpoint the specific points where optimizations could be needed. To determine the perfomance of individual parts of RedType a benchmark was created to isolate time spent on serialization, parsing, semantics and network communication to identify possible bottlenecks.

The method for the benchmark of RedType was a test environment, that is populated with needed test data for the operations, so the timing can be isolated. The core of the tests involve scaling the individual process to utilize more operations in a linear manner from one operation up to ten thousand. For example, testing the GET command would involve adding more GET operations to the script in a linear manner and not using loops. This test was completed 100 times and then the average was determined. These tests use a dedicated /debug/timing endpoint, which collects the timing data and then put into CSV files, which are used for outputting graphs afterwards. This approach allows for an overview of how specific parts of the system operate with an increasing load. The following will dive into the results from the GET operation tests.

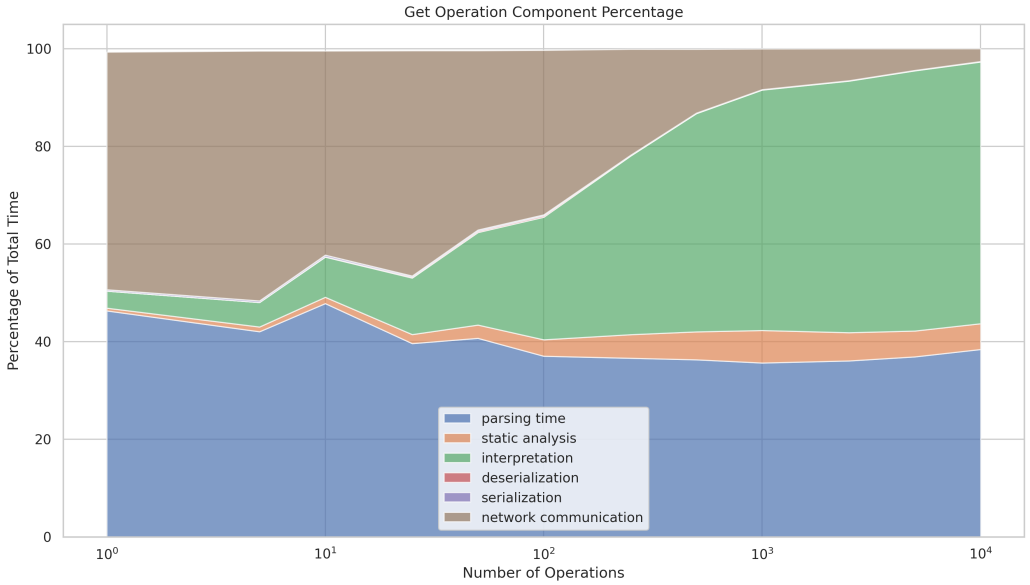

Figure 4: GET operations performance in percentage

Figure 4 above, shows how each component contribute to the total processing time of each query with the x-axis being the number of GET operations which increases exponentially. All data can be found in the csv file "get_scaling" found inside our RedType codebase. This graph illustrates a significant change in which component is the largest contributor to the total time. Queries with a very small number of operations spends a large part of its time on network communication and parsing. Roughly 40-45% of the total time is spent on network communication for these smaller requests, which is quite substantial. Similarly, parsing for the second largest amount of time in these smaller cases. However, as the number of operations within a single query increases, the cost of network communication decreases and ends with being only around 5-7% at ten thousand operations. While the percentage of time spent on parsing also decrease, this is not as dramatic, which indicates that the overhead does not scale proportionally. Instead, we can see that the interpretation time scales to become the biggest contributor, representing over 60% of the total time at ten thousand operations. In contrast, static analysis, deserialization, and serialization consistently maintain a minimal percentage of the total time.

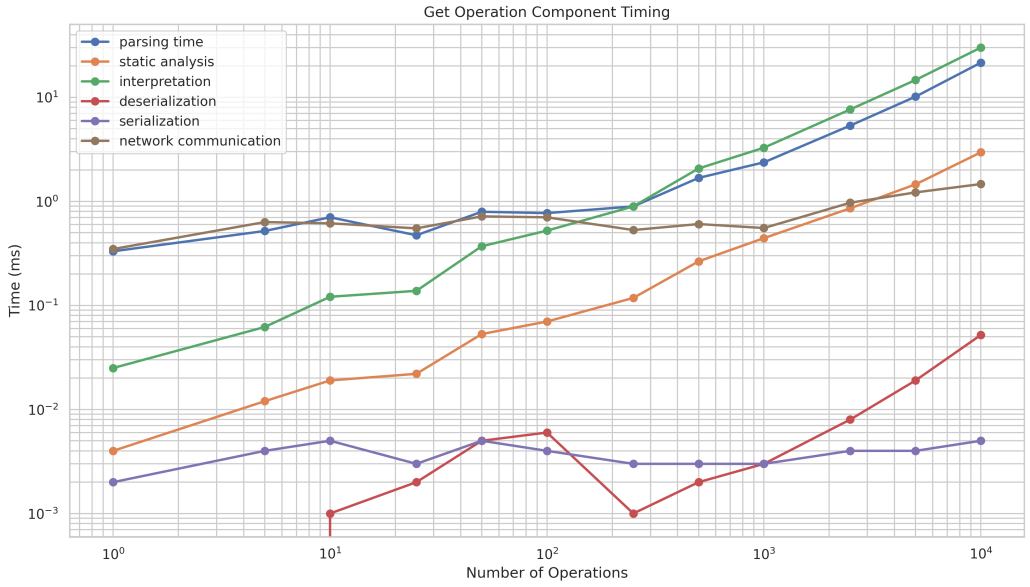

Figure 5: GET components scaling

To complement the percentage-based view, the graph in Figure 5, plots the absolute timings on a logarithmic scale. This showcases how each components timings increase. The interpretation time has a near linear increase in time spent as the number of operations increases, which would mean that the workload grows proportionally with the number of GETs executed within the query. Parsing time also increases, but is less intense and not

linear. Network communication time remains relatively flat and only increases slightly, which could be due to the sheer amount of data being transferred. This shows that the primary impact in time for network communication is a more fixed latency cost. Static analysis, deserialization, and serialization remain low even at scale, and are not significant bottlenecks.

Based on these observations, the interpretation and parsing are the primary bottleneck when queries involve a large number of operations. However, in smaller queries, the network overhead is quite a substantial part of the total time, which is important since smaller queries are a large part of the queries being sent to standard Redis servers. The highest priority for optimization therefore is both network overhead and parsing. Since parsing does not grow proportionally, this could indicate that there is a quite big fixed overhead for this work. In future work this should be investigated and optimized to lower the total time for each query. Network overhead should also be looked into, especially in terms of which protocol is best for this use case.

Concurrency performance & scaling benchmark

One of the major differences between Redis and the Redtype language is multi-threading. The following therefore aims to benchmark the theoretical performance increase from multi-threading our workload using Tokio as described in Section 8.1. This benchmark is against Redtype itself, unlike Section 10.4.

The test is similar to the ones in Section 10.4, but aim to increase the amount of available threads to Tokio. The methodology of the test is executed with a fixed client load that has 8 concurrent client threads, which each generate 100 asynchronous requests per iteration and a total of 20,000 operations per test configuration. The idea behind this is to isolate possible limitations on the server from client-side concurrency limitations by maintaining a consistent load during each test configuration. In total, 6 different test configurations were tested, where the amount of worker threads Tokio utilized was modified with the `TOKIO_WORKER_THREADS` environment variable. Each thread configuration (1, 2, 4, 8, 12 and 16) was tested 10 times to minimize impact of system noise and other variances. The test was executed on a single PC (acting as both client and server) with a i5-13500H Intel CPU. The amount of background processes were limited, but system noise is to be expected during the test. All testing results can be found under "benchmarks/concurrency_benchmark/benchmark_results" in the Redtype codebase.

The results of the tests reveal a significant performance improvement as server thread count increases, with improvements continuing even at higher thread counts. Performance across all three operations shows substantial improvement when scaling from 1 to 2 threads, with GET operations improving from 4,199 to 7,137 operations per second (OPS) (1.70x improvement), SET operations from 4,465 to 8,045 OPS (1.80x improvement),

and DEL operations from 4,327 to 7,639 OPS (1.77x improvement). Scaling to 4 threads results in an even bigger improvement, with GET rising to 16,387 OPS (2.30x increase from 2 threads), SET reaching 16,380 OPS (2.04x increase), and DEL reaching 16,523 OPS (2.16x increase). Further increases in thread count continue to yield significant improvements, particularly when scaling to 12 threads, as seen in Figure 6. At 16 threads, performance reaches its peak with GET operations at 26,250 OPS, SET operations at 27,418 OPS, and DEL operations at 26,838 OPS, representing overall improvements of 6.25x, 6.14x, and 6.20x respectively compared to single-threaded performance.

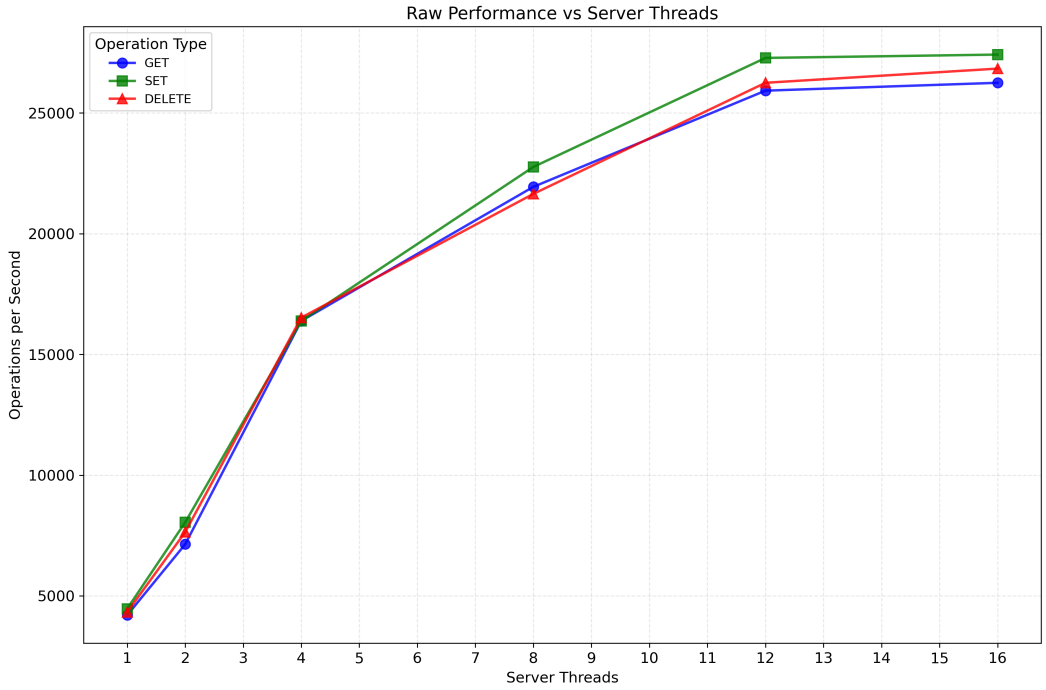

Figure 6: Operations per second versus server thread count

Figure 7 below, plots the observed performance speedup against a linear scaling scenario. Unlike the previous results, this graph showcases how the performance scales against a theoretical scenario. The scaling is particularly efficient up to 4 threads, being near-linear improvement, and continues to increase at higher thread counts, though with diminishing returns. This suggests that the Redtype server in large part utilizes additional CPU cores and that the architecture distributes workloads across available threads.

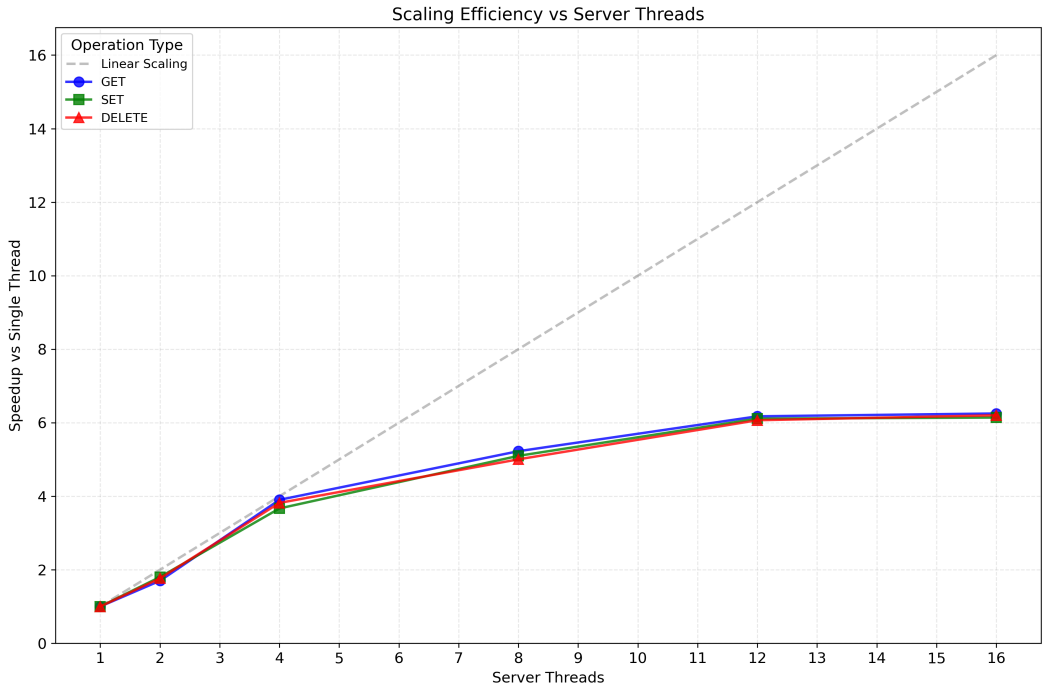

Figure 7: Scaling Efficiency vs Server Threads

11 Discussion

This section critically evaluates RedType's current design, its language features, and architectural decisions. We discuss the rationale behind core choices, analyze their impact, and identify areas for potential refinement.

11.1 Potential Syntax and Language Design Improvements

While RedType's current syntax is designed for clarity and typesafety, drawing inspiration from C-family languages, several enhancements could further improve the developer experience and align it more closely with common programming idioms. This section highlights a few areas where the language could be made more expressive and convenient.

One area for improvement is in looping constructs. Currently, RedType supports a for-each style loop, such as `for x in e`. While useful, the introduction of a C-style `for` loop, for instance, `for (x: Int = 0; x < 10; x = x + 1) { ... }`, would offer developers more fine-grained control over iteration. This is a familiar pattern for tasks requiring explicit indexing or more complex step logic. Similarly, adding a `do-while` loop would complement the existing `while` loop, providing a way to ensure the loop body executes at least once before the condition is checked. To make these loops even more versatile, formally specifying support for `break` and `continue` statements would allow for more flexible control flow within loop bodies, enabling early exits or skipping iterations as needed.

In terms of expressions and assignments, a couple of C-family staples could reduce verbosity. Enhanced assignment operators like `+=`, `-=`, and `*=` would provide a more concise way to write common update operations. Additionally, a ternary conditional operator, such as `result = condition ? valueIfTrue : valueIfFalse;`, would offer a compact alternative to simple `if-else` blocks for conditional assignments, improving readability for straightforward choices.

11.2 Language-Level Multithreading for Queries

Consideration was given to introducing language-level multithreading within RedType scripts. This would theoretically allow a single query to spawn multiple internal threads to execute parts of its logic (perhaps independent data-fetching operations) concurrently, with the main query thread collecting the results later. The appeal lies in potentially speeding up complex queries that involve several non-dependent operations.

However, implementing such a feature would introduce significant complexity and severe risks to RedType's core guarantees, particularly atomicity. If multiple threads within a single query operate on shared keys or data, race conditions could easily arise during the

execution of that one query. These internal race conditions would be incredibly difficult to manage and could lead to inconsistent states or compromised data, directly undermining the atomicity RedType strives to ensure. Given that atomicity is a fundamental principle of RedType's design for reliable concurrent operations, the potential for internal deadlocks and atomicity violations within a single query execution, makes language-level multithreading an unsuitable and overly complex feature for RedType. Therefore, it was decided that this capability would not be pursued.

11.3 LALRPOP Evaluation

The RedType project utilizes `LALRPOP` as its parser generator, primarily due to its compatibility with Rust and its support for LR(1) grammars. This made `LALRPOP` an attractive option for implementing a statically typed query language, as it integrates well with Rust's compile-time guarantees and provides a clear structure for grammar specification.

However, while `LALRPOP` was effective in enabling early progress and enforcing a strict grammar structure, it posed several challenges during the language's development. One notable limitation is its reliance on a single-token lookahead. This constraint made it difficult to express more complex or ambiguous grammar rules without introducing auxiliary productions or restructuring the grammar entirely. In certain cases, the inability to look ahead further did increase development overhead by requiring more careful rule design to avoid ambiguity.

Furthermore, debugging `LALRPOP` grammars can be unintuitive. The error messages often do not clearly indicate the root cause of a conflict, making it harder to iterate quickly when extending or modifying the language.

In retrospect, alternative parsing strategies such as Parsing Expression Grammars (PEG), might have offered greater flexibility. PEG parsers allow arbitrary lookahead and backtracking, which could have simplified grammar construction and improved error reporting. While PEG-based solutions can be slower in some cases due to backtracking, the trade-off could be worthwhile in scenarios where grammar complexity and developer productivity are higher priorities than raw parse-time performance.

Ultimately, `LALRPOP` served its purpose well for a language designed with static structure and performance in mind. However, for future iterations of RedType or more dynamically evolving languages, a PEG-based approach may be better suited to handle richer grammar features and more intuitive developer tooling.

11.4 Current Shortcomings of RedType

While RedType offers a novel combination of schema definition and query capabilities, there are still several criteria that make it less than ideal in its current form.

At the moment, there is no Language Server Protocol (LSP) support for RedType [20]. This absence significantly impacts the developer experience, as it removes quality-of-life features such as real-time error detection, type checking, auto-completion, and code formatting. Without these, writing and maintaining RedType code becomes more difficult and error-prone. In addition to this, RedType is currently embedded as a string inside e.g TypeScript code. This string-based approach means that even basic features like syntax highlighting are unavailable. Combined with the lack of LSP support, this makes the development process more tedious, slower, and generally suboptimal from a usability standpoint.

An important point is that RedType is slower than Redis, as shown in Section 10.4. For example, RedType does not outperform Redis in query execution. While RedType offers a more structured and type-safe query language, this advantage comes at the cost of runtime efficiency, making Redis the faster option for pure performance.

11.5 Test Evaluation

RedType's testing strategy was comprehensive, covering unit tests, integration testing, system-level validation, and performance benchmarking. The multi-layered approach successfully validated core functionality and correctness across all architectural components, from lexer and parser through static analysis to runtime behavior. Acceptance testing confirmed implementation of all "Must Have" MoSCoW requirements (see section 10.4), while exploratory testing with TypeScript clients uncovered real-world usage patterns and edge cases. However, it is important to note that testing was primarily conducted on personal computers, lacking the rigor of an isolated, standardized environment. For future, more robust validation, establishing dedicated servers for testing would mitigate inconsistencies and provide a more controlled setting.

Usability testing with external users was not conducted. This omission stems from several factors: the academic project's focus on technical feasibility over user experience, a limited timeline requiring prioritization of core language features, and RedType's current lack of developer-friendly tooling like LSP support and syntax highlighting. Additionally, meaningful acceptance testing is typically conducted by customers or end-users rather than the development team itself. The requirement to write RedType as plain strings within other languages creates a suboptimal development experience that would likely overshadow meaningful usability insights.

While this absence, along with the non-isolated testing environment, does not compromise the technical validity of RedType's implementation, it limits understanding of practical adoption potential and developer experience quality. For future iterations, usability studies evaluating learning curves, error message clarity, and productivity compared to alternatives like Redis with Lua scripting, conducted within a controlled server environment, would be invaluable.

Performance tests

Performance benchmarks reveal the trade-offs between RedType's type safety and execution speed (see section 10.4). RedType consistently underperforms Redis, being 1.91× slower for SET operations, 2.15× slower for GET operations, and 2.19× slower for DEL operations, as seen in table 3. Even for complex computational queries over 10,000 records, Redis maintains a 1.45× performance advantage, as seen in table 4. This gap reflects the overhead of RedType's type checking and validation mechanisms, contrasting with Redis's highly optimized C implementation that prioritizes raw performance over type safety guarantees.

Component analysis reveals specific bottlenecks within RedType's architecture. Network communication dominates smaller queries at 40-45% of execution time, seen in figure 4, representing fixed overhead that diminishes as query complexity increases. Parsing contributes significant overhead but shows fixed initialization costs rather than linear scaling. The primary bottleneck is interpretation, which scales from minimal impact in simple queries to over 60% of execution time at 10,000 operations. Static analysis, deserialization, and serialization remain consistently efficient across all scales. These findings indicate optimization priorities should focus on network protocols for small queries and interpreter efficiency for large operations.

In the concurrency performance tests under section 10.4, the scaling efficiency of the Redtype server is tested. It is observed in fig. 6 that the raw throughput reached approximately 27,000-28,000 at 16 server threads. The throughput scaled near-linearly up to 4 server threads with a 3.9x speedup as seen on fig. 7. Scaling continued but with diminishing returns, indicating a bottleneck within the system or testing environment as thread count increases.

Several factors may contribute to this observed performance ceiling. Potential resource contention within Redtype, possibly related to locking mechanisms, or other architectural limitations that manifest at higher concurrency levels. Furthermore, constraints imposed by the underlying hardware and test setup are also relevant. The tests were executed on a system equipped with an Intel i5-13500H processor (12 cores, 16 threads) [14], with both the client and server processes running on this single machine. This could likely influence the test results as the number of server threads approaches the physical limit, which also

could impact the client as they use the same system resources.

These possibly bottlenecks showcases the significance and lack of a isolated test environment. Currently, the client load configuration could impact the maximum server capacity and therefore scaling efficiency, which impacts the validity of the concurrency benchmarks. The most effective client load configuration was only achieved through experimental tuning of client-side parameters. In the future, the methodology should be improved to isolate the server from the client, which could help identify new bottlenecks. Moreover, the benchmarks were not tested on a cold cache which may affect initial query latency and memory related performance. Future benchmarks should include cold cache scenarios, to capture first access costs and better reflect real world performance. Additionally, simulating more realistic workloads could reveal further bottleneck scenarios, which are not clear in current tests.

11.6 Configuration Management and Project Workflow

Configuration control is important in this project to ensure consistency and traceability across development and design. The primary tool for collaborative document writing is Overleaf, mainly due to its version control and collaborative features. Visual aids are developed using Excalidraw for diagrams and Matplotlib for generating data-driven plots, which are integrated into the Overleaf document.

Visual Studio Code was used as the integrated development environment (IDE) for development within this project. The source code is centralized in a GitHub repository, where team members utilize either Git's CLI or the IDE's integrated functionality for version control operations. This approach to version control has served as a crucial tool for collaborative code development and as a backup mechanism. A key practice, enforced manually without a formal DevOps pipeline, is ensuring that if something fails in integration tests, it does not end up on the `main` branch. During or after development, the code can be built locally using Cargo, but a provided Docker Compose file also sets up the server environment and TypeScript interface, ensuring a consistent build process across different development environments.

Synchronization of Development and Documentation

A critical part of this project is designing and implementing a solution that is semantically correct. The correctness of this comes from the design, which raises the problem of maintaining synchronization between the codebase and documentation/design. This is especially true with multiple iterations that can change major parts of the program functionality. For this project, the group has used a discussion-based approach. When issues were identified, the group would coordinate updates in both code and report. Another

system was used to ensure consistency between development and documentation. Team members with knowledge about code would review report and vice-versa, to ensure that there was consistency between these components.

This process could be improved to increase the traceability of features and changes. Using a "feature branch" strategy in Git, where the code had to be reviewed by all teammates, would strengthen synchronization. This would be particularly vital if RedType were to be rolled out into production, where employing semantic versioning alongside such feature branching would be essential for managing releases and changes systematically. This could be further improved if the Latex report was built locally within the same Git environment. Then feature branches could include changes to both code and report and thereby show changes more easily. Another solution could be scheduled "sync meetings" that are dedicated to aligning the report with the current software state. Furthermore, while we have not implemented formal DevOps practices, future iterations would benefit significantly from them. For instance, a DevOps pipeline could automate the process of running integration tests and prevent pushes that do not pass these tests from being merged into the `main` branch. It could also automate the building of the RedType server, ensuring consistent and reliable deployments.

Communication and Task Management

Internal communication about development, task assignments, and general project discussions occurs in person or on Discord and Messenger, which encourages fast communication, though more informal. External communication, particularly with the project supervisor, is done via email.

Trello has been used for managing the timeline and tasks. This project employed an experimental and iterative workflow due to lack of experience with language design and the problem domain. Although the workflow was useful for revisions and changes based on new insights, it caused many problems with keeping a long-term time schedule.

In the future, a shorter and more agile iteration cycle with more focused planning could be used. The group could also try estimating the times for tasks, especially exploratory tasks, to shorten the delays and decide more quickly how to proceed.

Notetaking

Individual and group notes are written in Typst or Overleaf. The storing of files relevant to the project are stored in the integrated Overleaf report or distributed on Discord in terms of miscellaneous files. This could be improved by adopting a more centralized approach to file sharing.

12 Conclusion

This project took the challenge of implementing type safety and schema enforcement into in-memory key-value stores, an area where systems like Redis (particularly when using Lua scripting for complex operations) can encounter runtime type errors. We aimed to design RedType, a Redis-inspired system featuring a schema definition and a dedicated query language. The goal was to enforce type safety, ensure atomic query execution, support multithreading, and maintain in-memory performance, similar to Redis's, thereby offering a robust alternative to Redis with Lua for type-critical applications.

RedType provides a solution by embedding type safety and a robust concurrency model. Key design choices include a mandatory schema, a C-family inspired query language with static typing, Option types for handling absent data, and pessimistic locking with pre-declared keys to guarantee atomicity and prevent deadlocks.

Formal semantics were used in guiding the design and implementation of RedType. By defining static and dynamic semantics based on the abstract syntax, we formalized how RedType programs should behave, both in terms of type correctness and runtime execution. This formal foundation was essential for implementing schema-aware type checking and ensuring predictable query behavior.

Following the development of the semantics, RedType was constructed using a combination of key technologies to meet its design objectives. The implementation was done in Rust, leveraging the Tokio library to manage asynchronous task execution and handle concurrent queries efficiently. For the language implementation, LALRPOP was utilized to perform grammar-based parsing. The interpreter, designed in alignment with the formal semantics, included components for lexing, parsing, static analysis, and interpretation.

Testing confirmed RedType met all must-have requirements, including schema definition, server-side enforcement, a type safe query language, and atomic operations. Benchmarks against Redis with Lua showed that RedType is significantly slower for simple operations, and still lags behind in batch and computational queries, though to a lesser extent. These results highlight the trade-offs between type safety and raw speed. Through analysis, it was revealed that interpretation, parsing, and request handling contribute noticeably to execution overhead.

RedType has limitations, its pessimistic locking can be restrictive and impact throughput, but in spite of this, RedType successfully upholds its original problem statement:

How can an in-memory database system inspired by Redis be designed with a schema- and query language that enforces type safety before runtime, executes complex queries atomically, supports multi-threaded execution while still preserving the performance benefits of in-memory storage?

The implementation directly addresses limitations commonly encountered with Redis, particularly its reliance on Lua scripting for complex operations, which can compromise type safety and atomicity. Through extensive testing, RedType demonstrates that such a system is both practical and effective. While it does not yet match Redis in raw performance or tooling maturity, RedType establishes a robust foundation for applications where type correctness and data integrity are critical.

13 Future Work

While the current design establishes the core type safe language and basic operations for RedType, future iterations could explore various enhancements to improve performance, expressiveness, and operational capabilities. This section outlines potential directions for extending RedType beyond its initial scope, focusing on areas that could provide significant value to developers.

13.1 Advanced Data Structures

Right now, RedType mainly uses arrays to group data within scripts. These arrays require all items to be of the same type, and while this is useful, it means that functions can't easily return complex results made up of different kinds of data. A key area for future improvement is to add more advanced ways to structure data, similar to what Redis offers, such as Lists, Sets, HashMaps, and Sorted Lists (as mentioned in Requirement 9). Adding these would give developers better tools for working with data in their scripts.

13.2 Enhancing Code Clarity for Sequential Optional Operations

RedType's use of `Option<T>` types for operations that might not return a value, such as `GET` requests, is fundamental to its type safety. This design ensures that the potential absence of data is explicitly managed. However, when performing a sequence of such operations, where each step depends on the success of the previous one, developers must often write nested `match` statements. While this approach is robust, it can lead to code that is verbose, less immediately readable, and characterized by multiple levels of nesting.

Consider a common scenario where a function aims to retrieve several pieces of data sequentially. If a more direct chaining mechanism is not available, the code structure might resemble the following:

```
1  func getUserDisplayInfo_nested(userId: String): Option<String> {
2      userNameOpt: Option<String> = GET user[userId].name;
3
4      match userNameOpt {
5          Some(name) => {
6              userScoreOpt: Option<Int> = GET user[userId].score;
7
8              match userScoreOpt {
9                  Some(score) => {
10                     display: String = name + " - Score: " +
                         numericToString(score);
```

```
                    return Some(display); // Returning Some() is not
                    currently supported in RedType
                }
                None => {
                    return None;
                }
            }
        }
        None => {
            return None;
        }
    }
}
```

Listing 45: Nested `match` statements for sequential optional operations

This pattern, with its increasing indentation, can obscure the primary logic flow, especially in more complex queries.

To address this and improve the developer experience, a future enhancement for RedType could be the introduction of an "unwrap-or-return-None" operator, represented as a question mark `?`. This operator would offer a more concise way to chain operations that return `Option<T>` types.

The proposed `?` operator would function as follows: when applied to an expression evaluating to an `Option<T>` (for instance, `GET user[userId].name?`), it unwraps the value if it is `Some(v)`, making `v` the result of the `?` expression. However, if the expression evaluates to `None`, the `?` operator triggers an immediate return of `None` from the current function. Consequently, a key requirement for using this operator is that the enclosing function must itself be declared to return an `Option<U>` type (where `U` can be any type), thereby maintaining type consistency for these early `None` returns.

With such an operator, the previous example could be rewritten with significantly improved clarity:

```
func getUserDisplayInfo_improved(userId: String): Option<String> {
    name: String = GET user[userId].name?;
    score: Int = GET user[userId].score?;
    display: String = name + " - Score: " + numericToString(score);
    return Some(display);
}
```

Listing 46: Simplified code with the proposed `?` operator

This revised syntax effectively flattens the nested structure. The main sequence of operations becomes more apparent, and the propagation of a `None` result is handled implicitly and concisely by the `?` operator. This reduces boilerplate code while fully preserving the safety guarantees inherent in the `Option<T>` system. Introducing a `?` operator for handling sequential optional operations would therefore represent a valuable ergonomic enhancement to RedType. It would empower developers to write complex, type-safe queries with greater ease, resulting in code that is both more maintainable and more straightforward to understand.

13.3 Reducing Query Function Overhead

Currently, RedType queries require including the full function definition with each call. This leads to redundant data transmission, particularly for frequently used or complex functions. Future optimizations could reduce this overhead.

One approach is defining functions in a separate file, pushed to the server independently (akin to schemas). Queries could then invoke these registered functions by name, thereby eliminating repeated code transmission.

Alternatively, server-side function caching could be implemented. A function's code would be sent and cached on its first call within a session. Subsequent calls would reference the cached version, reducing data transfer without requiring a separate deployment step.

Both methods would decrease the data sent per query, improving submission speed and server efficiency, particularly for high-frequency operations.

13.4 Locking Entire Schema Types

RedType's current deadlock prevention strategy requires pre-declaring all specific keys a query will lock. This is effective when keys are known beforehand. However, situations arise where specific key identifiers are determined only at runtime, for instance, when a script processes a variable number of items based on its own intermediate calculations. In such cases, listing every potential key in advance becomes impractical.

Future work could explore extending the locking mechanism to support broader locks. For example, if a script needs to modify multiple instances of a `User` schema, instead of requiring `LOCK User[0], User[1], User[2];` and so on, RedType could allow locking the entire `User` schema type with a single command like `LOCK User;`. This would secure all keys associated with the `User` schema for the query's duration.

Allowing locks on entire schema types would offer developers more flexibility, especially when dealing with dynamically identified keys or operations affecting many items within

a schema. While this approach is simpler for such scenarios, it is a more encompassing lock than individual key locks and could reduce the potential for concurrent operations on other items within the same schema. Therefore, this broader locking would be a trade-off between convenience for certain use cases and the higher concurrency typically achieved with more specific key locking.

13.5 Client Communication Protocols and Data Formats

Efficient communication between the RedType server and its clients is important for overall system performance. Currently, RedType utilizes the HTTP protocol for client-server communication, with JSON as the data exchange format. A typical server response includes the operation's success status, a message, the returned values, and their corresponding RedType types, as illustrated below:

```
{
 "success": true,
 "message": "Command executed successfully",
 "values": {
   "result": "true"
 },
 "types": {
   "result": "bool"
 }
}
```

Listing 47: Example of a JSON response from the RedType server

While JSON is human-readable and widely supported, its text-based nature, including field names and structural characters (e.g., braces, quotes), means it consumes more bandwidth than more compact binary formats.

For future enhancements, optimizing data transfer could be beneficial. Redis, for instance, employs the Redis Serialization Protocol (RESP), a compact binary protocol designed for efficiency [38]. A future iteration of RedType could explore a custom, type-aware serialization format inspired by RESP to reduce bandwidth usage and potentially improve parsing speed. Additionally, while HTTP is robust, exploring protocols like WebSocket could offer lower-latency communication channels, which might be advantageous for certain RedType usage patterns.

13.6 Updating Schema

Currently, any modification to a RedType schema results in the complete deletion of all existing data associated with that schema. This approach simplifies the development process for this project phase, but it has significant limitations for real-world use.

For future enhancements, a more sophisticated schema migration strategy would be highly beneficial, especially if RedType instances manage large datasets. Such a system would involve comparing the previous and new schema definitions. When a field from the old schema is not found in the new one, the migration process could prompt the developer to specify if the field was intentionally removed or simply renamed, allowing data to be preserved under a new identifier. Fields explicitly removed would be dropped, while data for unchanged fields would be kept.

This kind of migration system could also incorporate advanced features like automatic type conversion for fields whose types have changed (e.g., converting stored integers to strings if a field's type changes from `Int` to `String`). Implementing this safely would require a well-defined process for developers to specify these conversion rules and renaming decisions, perhaps through an interactive command-line interface or a dedicated migration definition language.

The advantages of such a migration system are clear. If RedType were used as a primary database in a production environment, robust schema migrations would be essential to prevent data loss and ensure continuous operation during updates. Even if RedType is used as a cache, migrating existing cached data instead of deleting it entirely would save a significant number of cache misses. This would prevent the costly process of re-fetching or re-computing data from the underlying source each time the schema evolves.

For the scope of this project, the simpler approach of clearing the database upon schema changes was chosen to maintain focus on core language features. However, for a production-ready version of RedType, a comprehensive schema migration mechanism would be a critical addition.

13.7 Data-Persistence in RedType

Currently, when the RedType server is shut down or crashes, all the data stored in memory is erased. This is not optimal, especially for applications that rely on RedType as a persistent data store. Without any form of persistence, users risk complete data loss in the event of a failure or restart.

Consider a lightweight analytics system that records page view counts or user interactions in RedType. A sudden shutdown would result in the loss of all collected metrics since the

last restart, undermining the accuracy of reports and dashboards. In this case, persistent storage would ensure continuity and reliability.

As explained in the problem analysis, Redis offers two widely-used persistence strategies: Snapshotting (RDB) and Append-Only File (AOF).

Given RedType's use case, it would make sense to implement one of these persistence strategies or, as Redis allows for durability and data reliability. Doing so would transform RedType from a volatile, memory-only system into a more robust and production-capable storage solution.

Bibliography

[1] ANTLR. *ANTLR - ANother Tool for Language Recognition*. URL: https://www.antlr.org/.

[2] Apache Hudi Contributors. *Concurrency Control in Open Data Lakehouse - Apache Hudi*. https://hudi.apache.org/blog/2025/01/28/concurrency-control/. Accessed: 2025-05-15. 2025.

[3] E. G. Coffman Jr., M. J. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Computing Surveys* 3.2 (1971), pp. 67–78. DOI: 10.1145/356586.356588. URL: https://www.researchgate.net/publication/234803635_System_Deadlocks.

[4] nom contributors. *nom Rust crate documentation*. URL: https://docs.rs/nom/latest/nom/.

[5] Nick Desaulniers. *Rust: Pattern Matching and the Option Type*. 2013. URL: https://nickdesaulniers.github.io/blog/2013/05/07/rust-pattern-matching-and-the-option-type/.

[6] DragonflyDB. *Ensuring Atomicity: A Tale of Dragonfly Transactions*. 2025. URL: https://www.dragonflydb.io/blog/transactions-in-dragonfly.

[7] DragonflyDB. *Redis vs. Dragonfly Scalability and Performance*. 2025. URL: https://www.dragonflydb.io/blog/scaling-performance-redis-vs-dragonfly.

[8] Christopher Fanchi. "Redis: What It Is, What It Does, and Why You Should Care?" In: *backendless* (2022). URL: https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/.

[9] Martin Ganchev. *SQL is a Declarative Language*. 2023. URL: https://365datascience.com/tutorials/sql-tutorials/sql-declarative-language/.

[10] Vlad Gavriluk. "What is Rust and Why You Should Use It?" In: *arounda* (May 2024). URL: https://arounda.agency/blog/what-is-rust-and-why-you-should-use-it.

[11] GeeksforGeeks. *Database Recovery Techniques in DBMS*. https://www.geeksforgeeks.org/database-recovery-techniques-in-dbms/amp/. 2024.

[12] Hans Hüttel. *Transitions and Trees An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN: 978-0-521-14709-5.

[13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.4 Reference Manual*. [Online; accessed 26-May-2025]. Lua.org, PUC-Rio. 2024. URL: https://www.lua.org/manual/5.4/.

[14] Intel. *Intel® Core™ i5-13500H Processor*. Accessed: 2025-05-27. May 2025. URL: https://www.intel.com/content/www/us/en/products/sku/232147/intel-core-i513500h-processor-18m-cache-up-to-4-70-ghz/specifications.html.

[15] Will Johnston. *Redis as a Cache vs Redis as a Primary Database in 90 Seconds*. Aug. 2022. URL: https://redis.io/blog/redis-cache-vs-redis-primary-database-in-90-seconds/.

[16] Steve Klabnik, Carol Nichols, and The Rust Community. *The Rust Programming Language, Chapter 6.1: Defining an Enum*. 2023. URL: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html.

[17] Steve Klabnik et al. *The Rust Programming Language*. Version for Rust 1.85.0 (released 2025-02-17), edition = "2024". URL: https://doc.rust-lang.org/stable/book/index.html#the-rust-programming-language (visited on 05/19/2025).

[18] LALRPOP. *LALRPOP*. Nov. 2016. URL: https://github.com/lalrpop/lalrpop.

[19] LALRPOP. *The LALRPOP book*. Nov. 2016. URL: https://lalrpop.github.io/lalrpop/.

[20] Beyang Liu. *Sourcegraph, code intelligence, and the Language Server Protocol*. https://sourcegraph.com/blog/sourcegraph-code-intelligence-and-the-language-server-protocol. Accessed: 2025-05-26. 2017.

[21] Aditi Mishra. *Why is Redis So Fast Despite Being Single-Threaded?* Explains Redis's single-threaded event loop model, architecture, and performance characteristics. 2024. URL: https://medium.com/@aditimishra_541/why-is-redis-so-fast-despite-being-single-threaded-dc06ba33fc75 (visited on 09/15/2024).

[22] Torben Ægidius Mogensen. *Programming Language Design and Implementation*. Springwer, 2022.

[23] mozilla. "Handling text — strings in JavaScript". In: *mdn web docs* (2025). URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Strings.

[24] Eddy Nguyen. *Schema-driven development in 2021*. 2021. URL: https://99designs.com/blog/engineering/schema-driven-development/.

[25] Oracle Corporation. *MySQL 8.4 Reference Manual :: 17.6.6 Undo Logs*. https://dev.mysql.com/doc/refman/8.4/en/innodb-undo-logs.html. 2025.

[26] OSDev. *Deadlock - OSDev Wiki*. https://wiki.osdev.org/Deadlock. Version from 2007-01-10, Accessed: 2025-05-15. Content discusses deadlock prevention techniques including acquiring all resources at once. 2007.

[27] pest. *pest. The Elegant Parser*. URL: https://pest.rs/.

[28] Tutorials Point. *Compiler Design - Bottom-Up Parser*. Jan. 2025. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm.

[29] Redis. *Data structures*. URL: https://redis.io/technology/data-structures//.

[30] Redis. *Healthcare & EHR Database*. Describes Redis applications in healthcare for EHR systems, telemedicine platforms, patient monitoring, and healthcare analytics. 2024. URL: https://redis.io/industries/healthcare/ (visited on 11/20/2024).

[31] Redis. *How to use Redis for Query Caching*. Explains how Redis is used in e-commerce applications for product catalog caching, shopping cart management, and personalized recommendations. 2025. URL: https://redis.io/learn/howtos/solutions/microservices/caching (visited on 01/31/2025).

[32] Redis. *Introduction to Lua Scripting*. 2023. URL: https://redis.io/docs/manual/programmability/eval-intro/ (visited on 10/01/2023).
[33] Redis. *Lua Scripts & Timeouts*. Documents the limitations of Lua scripts in Redis, specifically the default execution time limit of 5 seconds and how scripts that exceed this limit are terminated to prevent blocking. 2023. URL: https://redis.io/docs/manual/programmability/lua-debugging/ (visited on 11/15/2023).
[34] Redis. *NoSQL Database*. URL: https://redis.io/nosql/what-is-nosql/.
[35] Redis. *Real-Time Fraud Detection*. Details how Redis is used in financial services for fraud detection, trading platforms, and payment processing with sub-millisecond latency. 2024. URL: https://redis.io/solutions/fraud-detection/ (visited on 11/20/2024).
[36] Redis. *Real-time leaderboard & ranking solutions*. Explains how Redis is used in gaming for leaderboards, player session data, and multiplayer game servers with real-time performance. 2024. URL: https://redis.io/solutions/leaderboards/ (visited on 11/20/2024).
[37] Redis. *Redis OM*. URL: https://www.npmjs.com/package/redis-om.
[38] Redis. *Redis serialization protocol specification*. https://redis.io/docs/latest/develop/reference/protocol-spec/. URL: https://redis.io/docs/latest/develop/reference/protocol-spec/ (visited on 05/20/2025).
[39] Redis. *Redis Transactions*. Explains the limitations of Redis transactions, including the lack of rollback capabilities when errors occur during execution. 2023. URL: https://redis.io/docs/manual/transactions/ (visited on 10/25/2023).
[40] Redis. *Transactions*. 2023. URL: https://redis.io/docs/manual/transactions/ (visited on 10/01/2023).
[41] Redis. *Writing Redis Scripts in Lua*. Describes how Lua scripts handle data types when interacting with Redis, highlighting the dynamic typing nature of Lua and the challenges for maintaining type safety in complex operations. 2023. URL: https://redis.io/docs/manual/programmability/lua-api/ (visited on 11/18/2023).
[42] sdfg610 (Sean). *Dims.kt*. 2025. URL: https://github.com/sdfg610/Dims.kt.
[43] *The Significance of Atomic Operations in Computer Science*. Mar. 2025. URL: https://startup-house.com/glossary/atomic-operation.
[44] Tokio. *Tokio Documentation*. URL: https://docs.rs/tokio/latest/tokio/.
[45] "What is Redis?" In: *IBM* (2025). URL: https://www.ibm.com/think/topics/redis.
[46] Wikipedia. *LL Parser*. URL: https://en.wikipedia.org/wiki/LL_parser.
[47] Wikipedia. *Parsing expression grammar*. URL: https://en.wikipedia.org/wiki/Parsing_expression_grammar.
[48] Wikipedia contributors. *Tony Hoare — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-May-2025]. 2025. URL: https://en.wikipedia.org/wiki/Tony_Hoare.

[49] Nkugwa Mark William. “WHAT ARE ATOMIC OPERATIONS IN SOFTWARE ENGINEERING - Nkugwa Mark William - Medium”. In: *Medium* (Oct. 2023). ISSN: 5581-7732. URL: https://nkugwamarkwilliam.medium.com/what-are-atomic-operations-in-software-engineering-e55c8177b3e2.

[50] Serdar Yegulalp. “What is Rust? Safe, fast, and easy software development”. In: *InfoWorld* (Nov. 2024). URL: https://www.infoworld.com/article/2255250/what-is-rust-safe-fast-and-easy-software-development.html.

A Query Examples

A.1 Query Example 2: Confirming a Purchase

```
LOCK Product["bf_special_item_001"].stockAvailable,
     Product["bf_special_item_001"].stockReserved;

func confirmPurchase(productId: String,
quantityPurchased: Int): String {

    stockReservedOpt: Option<Int> =
    GET Product[productId].stockReserved;
    currentStockReserved: Int = 0;

    match stockReservedOpt {
        Some(sr) => {
            currentStockReserved = sr;
        }
        None => {
            return "FAILURE: Product ID not found or reservation data error.";
        }
    }

    stockAvailableOpt: Option<Int> =
    GET Product[productId].stockAvailable;
    currentStockAvailable: Int = 0;

    match stockAvailableOpt {
        Some(sa) => {
            currentStockAvailable = sa;
        }
        None => {
            return "FAILURE: Product ID not found (stock available check).";
        }
    }

    if (quantityPurchased > 0 &&
        quantityPurchased <= currentStockReserved &&
        quantityPurchased <= currentStockAvailable) {

        DECR Product[productId].stockAvailable BY quantityPurchased;
        DECR Product[productId].stockReserved BY quantityPurchased;
        return "SUCCESS: Purchase confirmed and stock updated.";
    }
    else {
```

```
        if (quantityPurchased <= 0) {
            return "FAILURE: Quantity purchased must be positive.";
        }
        elif (quantityPurchased > currentStockReserved) {
            return "FAILURE: More items purchased than reserved.";
        }
        else {
            return "FAILURE: Critical stock discrepancy. Purchase cannot be
                confirmed.";
        }
    }
}

purchaseStatus: String =
confirmPurchase("bf_special_item_001", 1);

return purchaseStatus;
```

Listing 48: RedType Script: Confirm Purchase

A.2 Query Example 3: Restocking a Product

```
LOCK Product["bf_special_item_001"].stockAvailable;

func restockProduct(productId: String,
quantityAdded: Int): String {

    if (quantityAdded > 0) {
        INCR Product[productId].stockAvailable BY quantityAdded;
        return "SUCCESS: Product restocked.";
    }
    else {
        return "FAILURE: Quantity to add must be positive.";
    }
}

restockOutcome: String = restockProduct("bf_special_item_001", 1);

return restockOutcome;
```

Listing 49: RedType Script: Restock Product

A.3 Query Example 4: Getting a Stock Level Report

```
LOCK Product["bf_special_item_001"].stockAvailable,
     Product["bf_special_item_001"].stockReserved;

func getStockLevelReport(productId: String): String {

    stockAvailableOpt: Option<Int> = GET Product[productId].stockAvailable;
    currentStockAvailable: Int = 0;
    match stockAvailableOpt {
        Some(sa) => {
            currentStockAvailable = sa;
        }
        None => {
            return "PRODUCT_NOT_FOUND";
        }
    }

    stockReservedOpt: Option<Int> = GET Product[productId].stockReserved;
    currentStockReserved: Int = 0;
    match stockReservedOpt {
        Some(sr) => {
            currentStockReserved = sr;
        }
        None => {
            return "RESERVATION_DATA_ERROR";
        }
    }

    effectiveStock: Int = currentStockAvailable - currentStockReserved;
    if (effectiveStock >= 0) {
        return "Stock Available: " + numericToString(effectiveStock);
    }
    else {
        return "STOCK_DATA_INCONSISTENT";
    }
}

report: String = getStockLevelReport("bf_special_item_001");
return report;
```

Listing 50: RedType Script: Get Stock Report

B Built-in Array Functions

The core built-in functions for array manipulation include: *(Here, `T` stands for a simple type like `int` or `string`, making sure all items in the array are of this type.)*

- `len(arr: T[]): Int`
  - Returns how many items are in the array `arr`.
- `push(arr: T[], element: T)`
  - Adds an `element` to the end of the array `arr`. This changes `arr` directly.
- `pop(arr: T[]): Option<T>`
  - Removes and returns the last item from `arr`. This changes `arr` directly. It returns `None` if the array is empty, or `Some(item)` if it removed an item.
- `get(arr: T[], index: Int): Option<T>`
  - Gets the item at a specific `index` (position) from `arr`. It returns `None` if the `index` is not valid (e.g., too high or negative), or `Some(item)` if it finds an item.
- `insert(arr: T[], index: Int, element: T): Bool`
  - Puts an `element` into `arr` at a specific `index`. Items already at that position and after are shifted to make space. This changes `arr` directly. The script will return a Boolean depending on success, the boolean is false if one tries to insert out of bounds and true if not.
- `removeAt(arr: T[], index: Int): Option<T>`
  - Removes and returns the item at a specific `index` from `arr`. Items after that position are shifted to fill the gap. This changes `arr` directly. It returns `None` if the `index` is not valid, or `Some(item)` if it removed an item.

C Semantics

C.1 Static Semantics

C.1.1 Statements

$$[\text{VAR-DECL-INIT}_{TS}] \quad \frac{\Gamma' = \Gamma[x \mapsto T] \; \Gamma \vdash e : T}{\Gamma \vdash x : T = e : \texttt{ok}}$$

$$[\text{ASSIGN}_{TS}] \quad \frac{x \in \text{dom}(\Gamma) \quad \Gamma \vdash e : \Gamma(x)}{\Gamma \vdash x = e : \texttt{ok}}$$

$$[\text{ARRAY-ASSIGN}_{TS}] \quad \frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = T[] \quad \Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash x[e_1] = e_2 : \texttt{ok}}$$

$$[\text{IF-THEN-ELSE}_{TS}] \quad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{ok} \quad \Gamma \vdash S_2 : \texttt{ok}}{\Gamma \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 : \texttt{ok}}$$

$$[\text{IF-THEN}_{TS}] \quad \frac{\Gamma \vdash b : \texttt{bool} \quad \texttt{enter}(\Gamma) \vdash S : \texttt{ok}}{\Gamma \vdash \texttt{if } b \texttt{ then } S : \texttt{ok}}$$

$$[\text{FOR}_{TS}] \quad \frac{\Gamma \vdash e : T[] \quad \texttt{enter}(\Gamma)[x \mapsto T] \vdash S : \texttt{ok}}{\Gamma \vdash \texttt{for } x \texttt{ in } e \; S : \texttt{ok}}$$

$$[\text{WHILE}_{TS}] \quad \frac{\Gamma \vdash b : \texttt{bool} \quad \texttt{enter}(\Gamma) \vdash S : \texttt{ok}}{\Gamma \vdash \texttt{while } b \texttt{ do } S : \texttt{ok}}$$

$$[\text{SKIP}_{TS}] \quad \frac{}{\Gamma \vdash \texttt{skip} : \texttt{ok}}$$

$$[\text{RETURN-EXPR}_{TS}] \quad \frac{\Gamma \vdash e : T_{ret}, \text{ where } T_{ret} = \text{return type of the enclosing function}}{\Gamma, \pi \vdash \texttt{return } e : \texttt{ok}}$$

$$[\text{RETURN}_{TS}] \quad \frac{\text{(enclosing function has no declared return type)}}{\Gamma, \pi \; \vdash \; \texttt{return} : \texttt{ok}}$$

$$[\text{MATCH}_{TS}] \quad \frac{\begin{array}{l}\Gamma \vdash e : \texttt{Option}\langle T\rangle \\ \forall i \in 1..n,\ \big(\text{if } Case_i = \texttt{Some}(x_i) \Rightarrow S_i,\ \Gamma[x_i \mapsto T] \vdash S_i : \texttt{ok} \\ \quad \wedge\ \text{if } Case_i = \texttt{None} \Rightarrow S_i,\ \Gamma \vdash S_i : \texttt{ok}\big)\end{array}}{\Gamma \vdash \texttt{match}\ e\ \{\texttt{Case}_i\}_{i=1}^{n} : \texttt{ok}}$$

$$[\text{STMT-SEQ}_{TS}] \quad \frac{\forall i \in [1..n],\ \Gamma_{i-1}, \sigma_{i-1} \vdash S_i : \texttt{ok} \Rightarrow (\Gamma_i, \sigma_i)}{\Gamma_0, \sigma_0 \vdash \overrightarrow{S} : \texttt{ok} \Rightarrow (\Gamma_n, \sigma_n)} \quad \text{where } \overrightarrow{S} = S_1, S_2, \ldots, S_n$$

C.1.2 Database Operations

$$[\text{INCR}_{TS}] \quad \frac{\forall K_i \in \vec{K},\ \Gamma_s(K_i) \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma_s \vdash \texttt{INCR}\ \vec{K} : \texttt{ok}}$$

$$[\text{INCRBY-INT}_{TS}] \quad \frac{\Gamma_s, \sigma \vdash e \Rightarrow_{Exp} v \quad v \Rightarrow_{TS} \texttt{Int} \quad \forall K \in \vec{K},\ \Gamma_s(K_i) \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma_s, \sigma \vdash \texttt{INCR}\ \vec{K}\ \texttt{BY}\ e : \texttt{ok}}$$

$$[\text{INCRBY-DOUBLE}_{TS}] \quad \frac{\Gamma_s, \sigma \vdash e \Rightarrow_{Exp} v \quad v \Rightarrow_{TS} \{\texttt{Int}, \texttt{Double}\} \quad \forall K \in \vec{K},\ \Gamma_s(K_i) \in \{\texttt{Double}\}}{\Gamma_s, \sigma \vdash \texttt{INCR}\ \vec{K}\ \texttt{BY}\ e : \texttt{ok}}$$

$$[\text{SET}_{TS}] \quad \frac{\Gamma_s(K) = T \quad e : T}{\Gamma_s \vdash \texttt{SET}\ K = e : \texttt{ok}}$$

$$[\text{GET}_{TS}] \quad \frac{\Gamma_s(K) = T}{\Gamma_s,\ \Gamma \vdash x : \texttt{Option<}T\texttt{>} = \texttt{GET}\ K : \texttt{ok} \quad \text{and} \quad \Gamma' = \Gamma[x \mapsto \texttt{Option<}T\texttt{>}]}$$

$$[\text{GET-ASS}_{TS}] \quad \frac{\Gamma_s(K) = T \quad \Gamma(x) = \texttt{Option<}T\texttt{>}}{\Gamma_s,\ \Gamma \vdash x = \texttt{GET}\ K : \texttt{ok}}$$

$$[\text{DELETE}_{TS}] \quad \frac{\forall K_i \in \vec{K},\ K_i \in \text{dom}(\Gamma_s)}{\Gamma_s \vdash \texttt{DEL}\ \vec{K} : \texttt{ok}}$$

C.2 Dynamic Semantics

C.2.1 Program

$$[\text{PROGRAM}_{SSS}] \quad \frac{\Gamma_s \vdash \overrightarrow{S} \Rightarrow_{Schema} \Gamma_s' \quad \mathscr{L} \vdash \overrightarrow{L} \rightarrow \mathscr{L}' \quad \langle \overrightarrow{S_t}, \sigma, \mathscr{E}, \mathscr{L}', DB \rangle \Rightarrow (\sigma', \mathscr{E}', DB')}{\langle \overrightarrow{S} \ \overrightarrow{L} \ \overrightarrow{S_t}, \sigma, \mathscr{E}, \mathscr{L}, DB, \Gamma \rangle \Rightarrow_{Prog} (\sigma', \mathscr{E}', \mathscr{L}, DB', \Gamma')}$$

C.2.2 Locks

$$[\text{LOCK-DECL}_{SSS}] \quad \frac{\mathscr{L}(K_1) \neq \texttt{true} \quad \mathscr{L}' = \mathscr{L}[K_1 \mapsto \texttt{true}]}{\mathscr{L} \vdash \texttt{LOCK} \ K_1 :: K_{rest} \Rightarrow_{Lock} \langle \texttt{LOCK} \ K_{rest}, \mathscr{L}' \rangle}$$

C.2.3 Schema

$$[\text{SCHEMA-FIELD}_{BSS}] \quad \frac{}{\Gamma_s \vdash \{k : T\} \Rightarrow_{Schema} \{k \mapsto T\}}$$

$$[\text{SCHEMA-MULTI}_{BSS}] \quad \frac{\Gamma_s \vdash F \Rightarrow \Sigma_1 \quad \Gamma_s \vdash \{k : T\} \Rightarrow \Sigma_2}{\Gamma_s \vdash F \cup \{k : T\} \Rightarrow_{Schema} \Sigma_1 \cup \Sigma_2}$$

$$[\text{SCHEMA-DEF}_{BSS}] \quad \frac{\Sigma = \Gamma_s \vdash \{x_1 : T_1, \ldots, x_n : T_n\}}{\Gamma_s \vdash \mathrm{x}\{x_1 : T_1, \ldots, x_n : T_n\} \Rightarrow_{Schema} \Gamma_s[\mathrm{x} \mapsto \Sigma]}$$

C.2.4 Functions

$$[\text{FUNC-DEF}_{SSS}] \quad \frac{\pi' = \pi\big[f \mapsto \big((T_1, \ldots, T_n),\ T_{\text{ret}},\ S,\ [x_1, \ldots, x_n],\ \mathscr{E},\ \pi\big)\big]}{\mathscr{E}, \pi \ \vdash_l \ \big\langle \ \texttt{func} \ f(x_1 : T_1, \ldots, x_n : T_n) : T_{\text{ret}}\{S\},\ \sigma \big\rangle \ \rightarrow \ \langle \sigma, \pi' \rangle}$$

$$
\begin{array}{l}
\pi(f) = \big(S,\, [x_1, \ldots, x_n],\, \mathscr{E}',\, \pi'\big), \\
\forall i \in \{1..n\}.\ \mathscr{E}(y_i) = \ell_i,\ \sigma(\ell_i) = v_i, \\
\text{fresh locations } \ell_i' \text{ for each } i \text{ where } v_i \text{ is not an array}, \\
\mathscr{E}'' = \mathscr{E}' \left[x_i \mapsto \begin{cases} \ell_i & \text{if } v_i \text{ is an array} \\ \ell_i' & \text{otherwise} \end{cases} \;\middle|\; i \in \{1..n\} \right], \\
\sigma' = \sigma \left[\ell_i' \mapsto v_i \;\middle|\; v_i \text{ is not an array} \right], \\
\pi'' = \pi'[f \mapsto (S, [x_1, \ldots, x_n], \mathscr{E}', \pi')], \\
\dfrac{\mathscr{E}'', \pi'' \ \vdash_l \ \langle S, \sigma' \rangle \to \sigma''}{\mathscr{E}, \pi \ \vdash_l \ \langle f(y_1, \ldots, y_n), \sigma \rangle \to \sigma''}
\end{array}
\quad [\text{FUNC-CALL}_{SSS}]
$$

C.2.5 Statements

$$
[\text{DEC}_{SSS}] \quad \frac{x \notin \mathtt{dom}(\mathscr{E}) \quad \sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} v \quad \mathscr{E}' = \mathscr{E}[x \mapsto l]}{\mathscr{E} \vdash_l \langle x : T = e, \sigma, DB \rangle \Rightarrow_S (\sigma[l \mapsto v], \mathscr{E}', DB)}
$$

$$
[\text{ASS}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} v \quad \mathscr{E}(x) = l_x}{\mathscr{E} \vdash_l \langle x := e, \sigma, DB \rangle \Rightarrow_S (\sigma[l_x \mapsto v], DB)}
$$

$$
[\text{IF-ELSE-TRUE}_{SSS}] \quad \frac{\begin{array}{c} \sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} \mathtt{tt} \quad \mathscr{E}' = \mathtt{enter}(\mathscr{E}) \\ \mathscr{E}' \vdash_l \langle S_1, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')\mathtt{leave}(\mathscr{E}') \end{array}}{\mathscr{E} \vdash_l \langle \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')}
$$

$$
[\text{IF-ELSE-FALSE}_{SSS}] \quad \frac{\begin{array}{c} \sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} \mathtt{ff} \quad \mathscr{E}' = \mathtt{enter}(\mathscr{E}) \\ \mathscr{E}' \vdash_l \langle S_2, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')\mathtt{leave}(\mathscr{E}') \end{array}}{\mathscr{E} \vdash_l \langle \mathtt{if}\ e\ \mathtt{then}\ S_{t1}\ \mathtt{else}\ S_2, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')}
$$

$$
[\text{IF-TRUE}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} \mathtt{tt} \quad \mathscr{E}' = \mathtt{enter}(\mathscr{E}) \quad \mathscr{E}' \vdash_l \langle S_1, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')}{\mathscr{E} \vdash_l \langle \mathtt{if}\ e\ \mathtt{then}\ S_1, \sigma, DB \rangle \Rightarrow_S (\sigma', DB')}
$$

$$
[\text{IF-FALSE}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} \mathtt{ff}}{\mathscr{E} \vdash_l \langle \mathtt{if}\ e\ \mathtt{then}\ S_1, \sigma, DB \rangle \Rightarrow_S (\sigma, DB)}
$$

$$[\text{WHILE-TRUE}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \to_{\text{Exp}} \texttt{tt} \quad \mathscr{E}' = \texttt{enter}(\mathscr{E}) \quad \mathscr{E}' \vdash_l \langle S, \sigma, DB\rangle \Rightarrow_S (\sigma', DB') \quad \mathscr{E}'' = \texttt{leave}(\mathscr{E}')}{\mathscr{E} \vdash_l \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma, DB\rangle \Rightarrow_S \langle S; \texttt{while}\ e\ \texttt{do}\ S, \sigma', DB'\rangle}$$

$$[\text{WHILE-FALSE}_{SSS}] \quad \frac{\sigma \circ \mathscr{E} \vdash e \ \to_{\text{Exp}} \texttt{ff}}{\mathscr{E} \vdash_l \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma, DB\rangle \Rightarrow_S (\sigma, DB)}$$

$$[\text{MATCH-SOME}_{SSS}] \quad \frac{\mathscr{E}(x) = l_x \quad \sigma(l_x) = \texttt{Some}(v) \quad \mathscr{E}' = \mathscr{E}[y \mapsto l_1] \quad \sigma' = \sigma[l_1 \mapsto v] \quad \sigma', \mathscr{E}' \vdash_{l_2} \langle S_1, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}, DB'\rangle}{\sigma, \mathscr{E} \vdash_{l_1} \langle \texttt{match}\ x\ \texttt{with}\ \texttt{Some}(y)\ S_1\ \texttt{|}\ \texttt{None}\ S_2, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}'', DB'\rangle}$$

$$[\text{MATCH-NONE}_{SSS}] \quad \frac{\mathscr{E}(x) = l_x \quad \sigma(l_x) = \texttt{None} \quad \sigma, \mathscr{E} \vdash_{l_2} \langle S_2, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}'', DB'\rangle}{\sigma, \mathscr{E} \vdash_{l_1} \langle \texttt{match}\ x\ \texttt{with}\ \texttt{Some}(y)\ S_1\ \texttt{|}\ \texttt{None}\ S_2, DB\rangle \Rightarrow_S \langle \sigma'', \mathscr{E}, DB'\rangle}$$

C.2.6 Database Operations

$$[\text{DEL-STEP}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB' = \text{delete}(DB, K)}{\langle \texttt{DEL}\ \ K :: K_{\text{rest}},\ DB,\ \sigma,\ \mathscr{L}\rangle \ \Rightarrow_{Db}\ \langle \texttt{DEL}\ \ K_{\text{rest}},\ DB',\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{DEL-DONE}_{SSS}] \quad \frac{}{\langle \texttt{DEL}\ \ [\,],\ DB,\ \sigma,\ \mathscr{L}\rangle \ \Rightarrow\ \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{INCR-STEP}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) + 1]}{\langle \texttt{INCR}\ \ K :: K_{\text{rest}},\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow_{Db} \langle \texttt{INCR}\ \ K_{\text{rest}},\ DB',\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{INCR-DONE}_{SSS}] \quad \frac{}{\langle \texttt{INCR}\ \ [\,],\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{DECR-STEP}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) - 1]}{\langle \texttt{DECR}\ \ K :: K_{\text{rest}},\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow_{Db} \langle \texttt{DECR}\ \ K_{\text{rest}},\ DB',\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{DECR-DONE}_{SSS}] \quad \frac{}{\langle \texttt{DECR}\ [\,],\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{INCRBY-STEP}_{SSS}] \quad \frac{\sigma \vdash e \rightarrow_{\text{Aexp}} v \quad \mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) + v]}{\langle \texttt{INCR}\ K :: K_{\text{rest}}\ \texttt{BY}\ e,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{INCR}\ K_{\text{rest}}\ \texttt{BY}\ e,\ DB',\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{INCRBY-DONE}_{SSS}] \quad \frac{}{\langle \texttt{INCR}\ [\,]\ \texttt{BY}\ e,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{DECRBY-STEP}_{SSS}] \quad \frac{\sigma \vdash e \rightarrow_{\text{Aexp}} v \quad \mathscr{L}(K) = \texttt{true} \quad DB' = DB[K \mapsto DB(K) - v]}{\langle \texttt{DECR}\ K :: K_{\text{rest}}\ \texttt{BY}\ e,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{DECR}\ K_{\text{rest}}\ \texttt{BY}\ e,\ DB',\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{DECRBY-DONE}_{SSS}] \quad \frac{}{\langle \texttt{DECR}\ [\,]\ \texttt{BY}\ e,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma,\ \mathscr{L}\rangle}$$

$$[\text{SET}_{SSS}] \quad \frac{\sigma \vdash e \rightarrow_{\text{Aexp}} v \quad DB' = DB[K \mapsto v]}{\langle \texttt{SET}\ K\ \texttt{TO}\ e,\ DB,\ \sigma\rangle \Rightarrow \langle \texttt{skip},\ DB',\ \sigma\rangle}$$

$$[\text{GET}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB(K) = v}{\langle x = \texttt{GET}\ K,\ DB,\ \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma[x \mapsto \texttt{Some}(v)]\rangle}$$

$$[\text{GET-NONE}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad K \notin dom(DB)}{\langle x = \texttt{GET}\ K,\ DB,\ \sigma, \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma[x \mapsto \texttt{None}]\rangle}$$

$$[\text{GET-TYPED}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad DB(K) = v}{\langle x : \texttt{Option}\langle T\rangle = \texttt{GET}\ K,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma[x \mapsto \texttt{Some}(v)]\rangle}$$

$$[\text{GET-TYPED-NONE}_{SSS}] \quad \frac{\mathscr{L}(K) = \texttt{true} \quad K \notin \text{dom}(DB)}{\langle x : \texttt{Option}\langle T\rangle = \texttt{GET}\ K,\ DB,\ \sigma,\ \mathscr{L}\rangle \Rightarrow \langle \texttt{skip},\ DB,\ \sigma[x \mapsto \texttt{None}]\rangle}$$

D Transition Table

Table 5: Transition Systems for RedType Semantics

Name	$\mathcal{C}$ (Configuration)	$\rightarrow$ (Transition)	T (Terminal configuration)
Expression	$(\text{Expr} \cup \{tt, ff\} \cup \mathbb{Z} \cup \mathbb{R} \cup \text{String} \cup \text{Array}) \times \sigma \times \mathscr{E}$	$\rightarrow_{E_{\text{xp}}}$	$(\{tt, ff\} \cup \mathbb{Z} \cup \mathbb{R} \cup \text{String} \cup \text{Array}) \times \sigma \times \mathscr{E}$
Statements	$\text{Stmt} \times \sigma \times \mathscr{E} \times DB \times \pi \times v$	$\Rightarrow_S$	$(\text{Stmt} \times \sigma \times \mathscr{E} \times DB \times \pi) \cup (\sigma \times \mathscr{E} \times DB \times \pi) \cup (\sigma \times DB \times v \times \pi)$
Database Operations	$\text{DatabaseOp} \times DB \times \sigma \times \mathscr{L}$	$\Rightarrow_{\text{db}}$	$(\text{DatabaseOp} \times DB \times \sigma \times \mathscr{L}) \cup (DB \times \sigma \times \mathscr{L})$
Schema Definition	$\text{SchemaDefinition} \times \Gamma_s$	$\rightarrow_{\text{Schema}}$	Γ_s
Lock	$\texttt{Lock} \times \mathscr{L}$	$\Rightarrow_{\text{Lock}}$	$(\texttt{Lock} \times \mathscr{L}) \cup \mathscr{L}$
Program	$\text{Program} \times \sigma \times \mathscr{E} \times \mathscr{L} \times DB \times \Gamma_s$	$\Rightarrow_{\text{Prog}}$	$(\text{Program} \times \sigma \times \mathscr{E} \times \mathscr{L} \times DB \times \Gamma_s) \cup (\sigma' \times \mathscr{E}' \times \mathscr{L} \times DB' \times \Gamma_s')$

E How to use RedType

This section provides a practical guide to setting up and interacting with the RedType server and its accompanying client playground using Docker Compose. It outlines the necessary steps for getting the environment running and utilizing the demo environment for defining schemas and executing queries.

Before you begin, ensure that you have installed Docker and Docker Compose. If not already installed, Docker Desktop, which includes both Docker and Docker Compose, is recommended and can be downloaded from:
`https://docs.docker.com/desktop/`.

To get RedType up and running, follow these steps:

1. Download the `redtype-code.zip` file.
2. Unzip the `redtype-code.zip` file. This will create a folder containing the RedType server, the TypeScript client (including the playground), and a `compose.yaml` file configured to run both.
3. Navigate into the unzipped folder (e.g., `redtype-code`) in your preferred shell or terminal.
4. Execute the following command:

```
1 docker compose up
```

This command will build the necessary Docker images and start both the RedType server and the RedType client playground. The RedType server will be operating internally, and the client playground will be accessible by navigating to `http://localhost:3000` in your web browser on the same machine.

With the RedType server and client playground running, let us proceed to interact with it.

E.1 Using the RedType Client Demo Playground

The RedType Client Demo playground provides an interactive way to test RedType. Once you have accessed `http://localhost:3000` in your browser, you should see the demo page as illustrated in Figure 8.

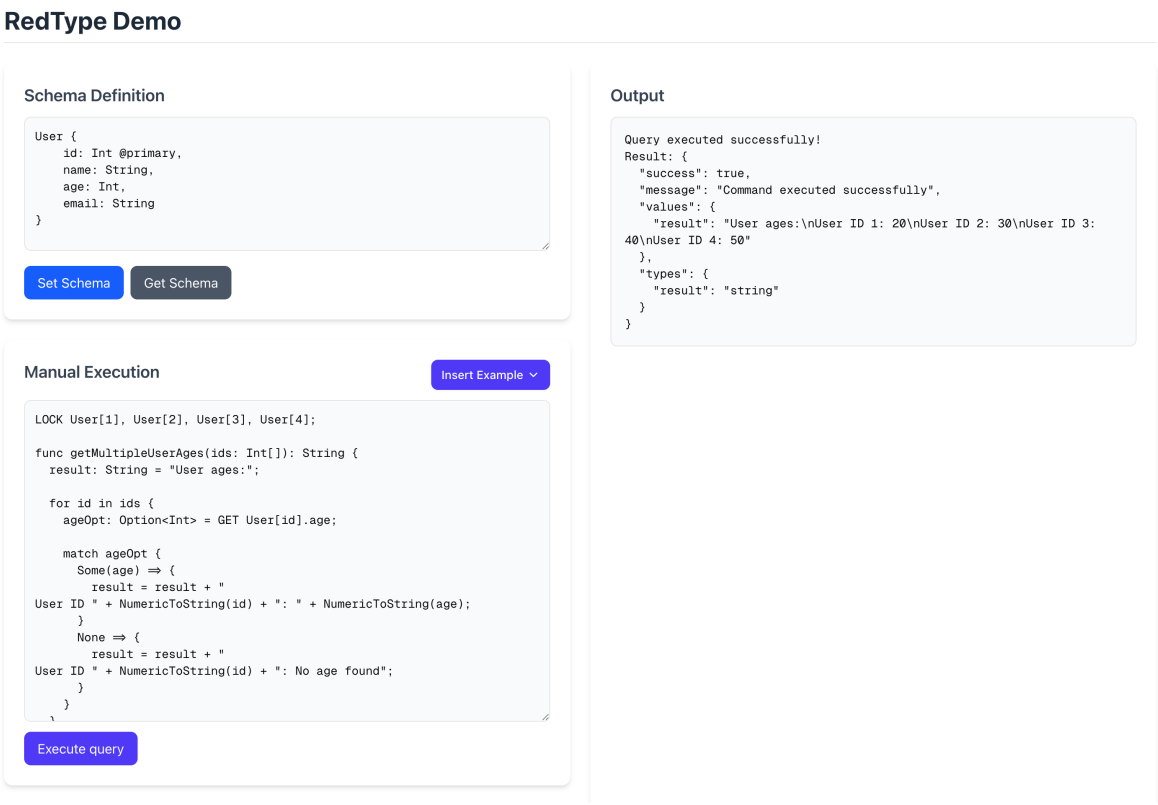

Figure 8: RedType Client Demo page

The demo page is divided into two main sections: the left side allows you to execute operations, while the right side displays the input and corresponding JSON responses.

In the "Schema" box, you can define or retrieve the current schema. The server's response to these actions will be displayed on the right.

Beneath the schema management area is a box for the "Manual Execution of Queries". In the top right corner of this box, you can click "Insert Example" to populate the input field with various sample queries. This feature allows you to explore different RedType functionalities, such as defining data structures, mutating data, and retrieving it.

You have now successfully configured RedType using Docker Compose and learned how to interact with it via the client playground.

The provided `typescript-client` (part of the `redtype-code.zip`) serves as a reference client implementation. Should you wish to use RedType with other programming languages, a corresponding client library would need to be implemented, interacting with the RedType server API (as the TypeScript client does).

This completes the guide on how to set up and begin using RedType.

F LALRPOP Grammar & ast.rs

F.1 LALRPOP Grammar

```
use crate::parsers::query::ast::*;

grammar;

// ===== TERMINALS =====
pub IntLiteral: i64 = <s:r"-?(?:[0-9]+)"> => s.parse().unwrap();
pub Float: f64 = <s:r"-?(?:[0-9]+\.[0-9]+)"> => s.parse().unwrap();
pub Ident: String = <s:r"[a-zA-Z_][a-zA-Z0-9_]*"> => s.to_string();
pub StringLiteral: String = <s:r#""[^"]*""#> => s[1..s.len()-1].to_string(); // remove quotes after parsing

// Arithmetic tokens
Plus: () = "+";
Minus: () = "-";
Star: () = "*";
Slash: () = "/";
Percent: () = "%";
Caret: () = "^";

// general tokens
LParen: () = "(";
RParen: () = ")";
LBrack: () = "[";
RBrack: () = "]";
Lt: () = "<";
Gt: () = ">";
CommaLit: () = ",";

// Boolean tokens
EqualEqual: () = "==";
NotEqual: () = "!=";
LessThan: () = "<";
LessEqual: () = "<=";
GreaterThan: () = ">";
GreaterEqual: () = ">=";
Excl: () = "!";
AndAnd: () = "&&";
OrOr: () = "||";

// Type tokens
Int: () = "Int";
```

```
Double: () = "Double";
String: () = "String";
Bool: () = "Bool";

// ===== RedType =====
pub RedType: RedType = {
    <schemadef:SchemaDefinition> => RedType {
        schemadef: Some(schemadef),
        program: None
    },
    <program:Program> => RedType {
        schemadef: None,
        program: Some(program)
    }
}

pub Program: Program = {
    <lock:Lock?> <statements:Stmt*> => Program {
        lock,
        statements
    }
}

// ===== SCHEMA =====
pub SchemaDefinition: SchemaDefinition = {
    <typedefs:Comma<TypeDefinition>> => SchemaDefinition(typedefs)
};

TypeDefinition: TypeDefinition = {
    <name:TypeName> "{" <fields:Comma<FieldDefinition>> "}" => TypeDefinition {
        name,
        fields
    }
};

FieldDefinition: FieldDefinition = {
    <name:FieldName> ":" <ty:Type> <constraint:Constraint?> => FieldDefinition {
        name,
        ty,
        constraint
    }
};

Constraint: Constraint = {
```

```
85      "@primary" => Constraint::Primary,
86  };
87
88  TypeName: String = Ident;
89  FieldName: String = Ident;
90
91  // ===== LOCK =====
92  pub Lock: Lock = {
93      "LOCK" <keys:Comma<KeyIdentifier>> ";" => Lock { keys }
94  };
95
96
97  // ===== STATEMENTS =====
98
99  pub Stmt: Stmt = {
100     #[precedence(level="0")]
101     // Declarations and assignments
102     <id:Ident> ":" <ty:Type> "=" <expr:Expr> ";" => Stmt::Declare(id.into(), ty,
            expr),
103     <id:Ident> "=" <expr:Expr> ";" => Stmt::Assign(id.into(), expr),
104     //<id:Ident> "[" <index:Expr> "]" "=" <expr:Expr> ";" =>
            Stmt::ArrayAssign(id.into(), index, expr),
105
106     #[precedence(level="1")]
107     <expr:Expr> ";" => Stmt::Expr(expr),
108
109     // Function definitions
110     FunctionDef => Stmt::FunctionDef(<>),
111
112     // Control flow
113     IfStmt,
114     ForLoop,
115     WhileLoop,
116
117     // Database operations
118     <dbop:DatabaseOp> ";" => Stmt::DatabaseOp(dbop),
119
120     // Special statements
121     "skip" ";" => Stmt::Skip,
122     "return" <expr:Expr?> ";" => Stmt::Return(expr),
123     "match" <expr:Expr> "{" <cases:Case*> "}" => Stmt::Match(expr, cases)
124 };
125
126 FunctionDef: FunctionDef = {
```

```
    "func" <name:Ident> "(" <params:ParamList?> ")" <ret:ReturnType?>
    "{" <body:Stmt*> "}" => FunctionDef {
        name: name.into(),
        params: params.unwrap_or_else(|| vec![]),
        return_type: ret,
        body: body
    }
};

ParamList: Vec<Param> = {
    Comma<Param>
};

Param: Param = {
    <id:Ident> ":" <ty:Type> => Param {
        name: id.into(),
        ty: ty
    }
};

ReturnType: Type = {
    ":" <ty:Type> => ty
};

IfStmt: Stmt = {
    "if" "(" <cond:OrExpression> ")" "{" <then:Stmt*> "}"
    <elifs:ElifBranch*>
    <else_branch:ElseBranch?> => Stmt::If {
        cond,
        then: then,
        elif_branches: elifs,
        else_branch: else_branch
    }
};

ElifBranch: (Expr, Vec<Stmt>) = {
    "elif" "(" <cond:OrExpression> ")" "{" <body:Stmt*> "}" => (cond, body)
};

ElseBranch: Vec<Stmt> = {
    "else" "{" <body:Stmt*> "}" => body
};

ForLoop: Stmt = {
```

```
171     "for" <id:Ident> "in" <iter:Expr> "{" <body:Stmt*> "}" =>
172     Stmt::For(id.into(), iter, body)
173 };
174 
175 WhileLoop: Stmt = {
176     "while" "(" <cond:OrExpression> ")" "do" "{" <body:Stmt*> "}" =>
177     Stmt::While(cond, body)
178 };
179 
180 Case: Case = {
181     "Some" "(" <identifier:Ident> ")" "=>" "{" <body:Stmt*> "}" =>
            Case::SomeCase(identifier, body),
182     "None" "=>" "{" <body:Stmt*> "}" => Case::NoneCase(body)
183 };
184 
185 // ===== DatabaseOp =====
186 pub DatabaseOp: DatabaseOp = {
187     "DEL" <ids:Comma<KeyIdentifier>> => DatabaseOp::Delete(ids.into_iter().map(|s|
            s.into()).collect()),
188     "INCR" <ids:Comma<KeyIdentifier>> <by:("BY" Aexp)?> => DatabaseOp::Incr(
189         ids.into_iter().map(|s| s.into()).collect(),
190         by.map(|(_, aexp)| aexp)
191     ),
192     "DECR" <ids:Comma<KeyIdentifier>> <by:("BY" Aexp)?> => DatabaseOp::Decr(
193         ids.into_iter().map(|s| s.into()).collect(),
194         by.map(|(_, aexp)| aexp)
195     ),
196     "SET" <id:KeyIdentifier> "TO" <expr:Expr> => DatabaseOp::Set(id.into(), expr),
197     <id1:Ident> ":" <ty:Type> "=" "GET" <id2:KeyIdentifier> =>
            DatabaseOp::Get(id1.into(), id2.into(), Some(ty)), // with assignment of
            type
198     <id1:Ident> "=" "GET" <id2:KeyIdentifier> => DatabaseOp::Get(id1.into(),
            id2.into(), None), // without assignment of type
199 };
200 
201 KeyIdentifier: KeyIdentifier = {
202     <type_name:TypeName> LBrack <key_value:Expr> RBrack <field_name:("." Ident)?>
            => KeyIdentifier {
203         type_name,
204         key_value,
205         field_name: field_name.map(|(_, name)| name) // remove the dot
206     }
207 };
208 // ===== TYPES =====
```

```
pub Type: Type = {
    CoreType => Type::CoreType(<>),
    OptionType,
};

pub CoreType: CoreType = {
    PrimitiveType => CoreType::Primitive(<>),
    CollectionType => CoreType::Array(<>),
};

pub OptionType: Type = {
    "Option" "<" <ty:CoreType> ">" => Type::Option(Box::new(ty))
};

pub PrimitiveType: PrimitiveType = {
    Int => PrimitiveType::Int,
    Double => PrimitiveType::Double,
    String => PrimitiveType::String,
    Bool => PrimitiveType::Bool,
};

pub CollectionType: CollectionType = {
    <ty:PrimitiveType> LBrack RBrack => CollectionType::Array(ty),
}

// ===== EXPRESSIONS =====
pub Expr: Expr = {
    OrExpression,
};

pub OrExpression: Expr = {
    <left:OrExpression> OrOr <right:AndExpression> => Expr::BinOp {
        left: Box::new(left),
        op: BinOp::Or,
        right: Box::new(right),
    },
    AndExpression
};

pub AndExpression: Expr = {
    <left:AndExpression> AndAnd <right:CmpExpression> => Expr::BinOp {
        left: Box::new(left),
        op: BinOp::And,
        right: Box::new(right),
```

```
253     },
254     CmpExpression
255
256 };
257
258 pub CmpExpression: Expr = {
259     <left:Aexp> <op:RelOp> <right:Aexp> => Expr::Compare {
260         left: Box::new(left),
261         op,
262         right: Box::new(right),
263     },
264     Aexp
265 };
266
267 pub Aexp: Expr = {
268     <left:Aexp> <op:AddOp> <right:Term> => Expr::BinOp {
269         left: Box::new(left),
270         op,
271         right: Box::new(right),
272     },
273     Term
274 };
275
276 pub AddOp: BinOp = {
277     "+" => BinOp::Add,
278     "-" => BinOp::Sub,
279 };
280
281 pub Term: Expr = {
282     <left:Term> <op:MulOp> <right:Factor> => Expr::BinOp {
283         left: Box::new(left),
284         op,
285         right: Box::new(right),
286     },
287     Factor
288 };
289
290 pub MulOp: BinOp = {
291     "*" => BinOp::Mul,
292     "/" => BinOp::Div,
293     "%" => BinOp::Mod,
294 };
295
296 pub Factor: Expr = {
```

```
297     <op:UnaryOp> <e:Power> => Expr::UnaryOp { op, expr: Box::new(e) },
298     Power
299 };
300
301
302 pub UnaryOp: UnaryOp = {
303     "+" => UnaryOp::Pos,
304     "-" => UnaryOp::Neg,
305     "!" => UnaryOp::Excl
306 }
307
308 pub Power: Expr = {
309     <left:AtomExpr> "^" <right:Power> => Expr::BinOp {
310         left: Box::new(left),
311         op: BinOp::Pow,
312         right: Box::new(right),
313     },
314     AtomExpr
315 };
316
317 pub AtomExpr: Expr = {
318     Atom,
319     <f:Ident> "(" <args:ArgList?> ")" => Expr::Call(f.into(),
            args.unwrap_or_default()),
320     LBrack <items:ExprList?> RBrack => Expr::Array(items.unwrap_or_default()),
321     //<id:Ident> LBrack <index:Expr> RBrack => Expr::ArrayAccess(id.into(),
            Box::new(index)),
322     "Some" "(" <expr:Expr> ")" => Expr::SomeExpr(Box::new(expr)),
323     "None" => Expr::NoneExpr,
324 };
325
326 pub Atom: Expr = {
327     StringLiteral => Expr::Constant(Constant::String(<>)),
328     IntLiteral => Expr::Constant(Constant::Int(<>)),
329     Float => Expr::Constant(Constant::Double(<>)),
330     <id:Ident> => Expr::Identifier(id),
331     "(" <e:Expr> ")" => e,
332     "true" => Expr::Constant(Constant::Bool(true)),
333     "false" => Expr::Constant(Constant::Bool(false))
334 };
335
336 RelOp: RelOp = {
337     LessThan => RelOp::Lt,
338     LessEqual => RelOp::Le,
```

```
    GreaterThan => RelOp::Gt,
    GreaterEqual => RelOp::Ge,
    EqualEqual => RelOp::Eq,
    NotEqual => RelOp::Neq,
};


// ===== GENERAL =====
pub ArgList: Vec<Expr> = {
    Comma<Expr>
};

pub ExprList: Vec<Expr> = {
    Comma<Expr>
};

Comma<T>: Vec<T> = {
    <first:T> <rest:("," T)*> => {
        let mut values = vec![first];
        for (_, val) in rest {
            values.push(val);
        }
        values
    }
};

Pipe<T>: Vec<T> = {
    <first:T> <rest:("|" T)*> => {
        let mut values = vec![first];
        for (_, val) in rest {
            values.push(val);
        }
        values
    }
};
```

F.2 ast.rs

```
#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct RedType {
    pub schemadef: Option<SchemaDefinition>,
    pub program: Option<Program>,
}
#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct Program {
    pub lock: Option<Lock>,
    pub statements: Vec<Stmt>,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct SchemaDefinition(pub Vec<TypeDefinition>);

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct TypeDefinition {
    pub name: String,
    pub fields: Vec<FieldDefinition>,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct FieldDefinition {
    pub name: String,
    pub ty: Type,
    pub constraint: Option<Constraint>,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum Constraint {
    Primary,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct Lock {
    pub keys: Vec<KeyIdentifier>,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct FunctionDef {
    pub name: String,
    pub params: Vec<Param>,
    pub return_type: Option<Type>,
    pub body: Vec<Stmt>,
```

```
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct Param {
    pub name: String,
    pub ty: Type,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum Stmt {
    FunctionDef(FunctionDef),
    Declare(String, Type, Expr),
    Assign(String, Expr),
    //ArrayAssign(String, Expr, Expr),
    Expr(Expr),
    If {
        cond: Expr,
        then: Vec<Stmt>,
        elif_branches: Vec<(Expr, Vec<Stmt>)>,
        else_branch: Option<Vec<Stmt>>,
    },
    For(String, Expr, Vec<Stmt>),
    While(Expr, Vec<Stmt>),
    DatabaseOp(DatabaseOp),
    Skip,
    Return(Option<Expr>),
    Match(Expr, Vec<Case>),
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum Case {
    SomeCase(String, Vec<Stmt>),
    NoneCase(Vec<Stmt>),
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum DatabaseOp {
    Delete(Vec<KeyIdentifier>),
    Incr(Vec<KeyIdentifier>, Option<Expr>),
    Decr(Vec<KeyIdentifier>, Option<Expr>),
    Set(KeyIdentifier, Expr),
    Get(String, KeyIdentifier, Option<Type>),
}

```

```rust
#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub struct KeyIdentifier {
    pub type_name: String,
    pub key_value: Expr,
    pub field_name: Option<String>,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum Type {
    CoreType(CoreType),
    Option(Box<CoreType>),
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum CoreType {
    Primitive(PrimitiveType),
    Array(CollectionType),
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum CollectionType {
    Array(PrimitiveType),
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum PrimitiveType {
    Int,
    Double,
    String,
    Bool,
}

#[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum Expr {
    BinOp {
        left: Box<Expr>,
        op: BinOp,
        right: Box<Expr>,
    },
    UnaryOp {
        op: UnaryOp,
        expr: Box<Expr>,
    },
    Compare {
```

```
132         left: Box<Expr>,
133         op: RelOp,
134         right: Box<Expr>,
135     },
136     SomeExpr(Box<Expr>),
137     NoneExpr,
138     Constant(Constant),
139     Identifier(String),
140     Call(String, Vec<Expr>),
141     Array(Vec<Expr>),
142     //ArrayAccess(String, Box<Expr>),
143 }
144 
145 #[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
146 pub enum Constant {
147     Int(i64),
148     Double(f64),
149     String(String),
150     Bool(bool),
151 }
152 
153 #[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
154 pub enum BinOp {
155     Add,
156     Sub,
157     Mul,
158     Div,
159     Pow,
160     Mod,
161     And,
162     Or,
163 }
164 
165 #[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
166 pub enum UnaryOp {
167     Pos,
168     Neg,
169     Excl,
170 }
171 
172 #[derive(Debug, Clone, PartialEq, serde::Serialize, serde::Deserialize)]
173 pub enum RelOp {
174     Eq,
175     Neq,
```

```
    Gt,
    Ge,
    Lt,
    Le,
}
```