McMaster University

Snapshot Relational Service in Haskell with Servant and SQLite

Author: Yi Lai

Supervisor: Prof. Wolfram Kahl

July 13, 2025

Contents

Abstract

This report presents the design, implementation, and evaluation of a multi-relational snapshot server based on Haskell, Servant, and SQLite. It introduces the motivation, technical contributions, and discusses limitations and future directions.

1 Introduction

In recent years, the use of relations and relation algebra has become an important didactic tool for teaching logic, specification, and formal reasoning. However, effective teaching requires not only abstract concepts but also concrete and consistent examples that students can manipulate. A typical challenge arises when examples are drawn from dynamic, real-world datasets such as Wikidata: while these datasets are rich and continuously updated, they also evolve rapidly, which makes it difficult to ensure that all students are working with the same version of the data. If, for instance, a new subway station is opened or a new member is added to a royal family, the underlying relation changes, and different students may inadvertently be working on different tasks.

This project addresses that challenge by implementing a *multi-relational snapshot server* that provides consistent and reproducible relational datasets for teaching and experimentation. The core idea is to maintain *snapshots* of real-world relations (e.g., subway reachability or ancestry) that can be queried, cached, and shared with students. By working on explicitly defined snapshots, instructors ensure that all students reason over the same data, while still preserving a connection to realistic domains.

The server operates on two complementary levels:

- **Administration and snapshot generation.** Administrators define SPARQL-like queries against Wikidata (or other external sources). The server evaluates these queries and stores the resulting relations as structured snapshots in a local database.
- **Snapshot serving and student use.** Once generated, snapshots are exposed through a RESTful HTTP API, a command-line interface (CLI), and a lightweight browser-based UI. Students can consume the cached relations directly, for example to evaluate relation-algebraic expressions (e.g., in CalcCheck), without depending on the live state of external services.

The system is implemented in **Haskell**, using **Servant** to specify and expose the API at the type level, and **SQLite** as a lightweight relational backend for storing snapshot data. This stack emphasizes clarity, modularity, and type safety: Servant ties the API specification to its implementation, and Haskell's strong typing provides reliable abstractions for modeling relational data. Snapshots are represented as JSON-serializable structures, enabling straightforward storage, retrieval, and exchange.

The project makes the following contributions:

- **Architecture and Implementation:** A modular snapshot server with clear API, server logic, database, and model layers in a Haskell + Servant + SQLite stack.
- **SPARQL-inspired Query Interface:** An interface for defining SPARQL-like queries that can be transformed into local snapshots, bridging external knowledge sources with stable relational datasets.
- **Multiple Interaction Modes:** A REST API, a CLI tool, and a browser-based UI to support diverse workflows for administrators and students.
- **Demonstration and Testing:** Construction of example snapshots and validation via API calls, CLI operations, and browser-based visualization to demonstrate correct storage, retrieval, and use.

By decoupling snapshot generation from snapshot consumption, the server reconciles the dynamism of real-world knowledge bases with the pedagogical need for reproducibility. Instructors

can curate interesting example relations from Wikidata, freeze them into snapshots for a course offering, and later regenerate or extend them for subsequent offerings.

Chapter 2 surveys background and related work (graph data, snapshots, SPARQL, and lightweight data services in Haskell). Chapter 3 presents the overall system architecture and dataflow. Chapter 4 details the implementation of each module. Chapter 5 evaluates functionality through concrete examples and tests. Chapter 6 discusses strengths, limitations, and potential improvements. Chapter 7 concludes.

2 Background and Related Work

This chapter surveys the key concepts and related systems that provide the context for our multi-relational snapshot server. We begin with graph-structured data and snapshots, then review SPARQL and the Wikidata ecosystem, and finally compare related platforms and tools such as YAGO, RDF triple stores, and graph databases.

2.1 Graph Data and Snapshots

Graph-structured data has become a central paradigm for representing knowledge and relationships between entities. Unlike relational tables, graph data models emphasize *nodes* (entities) and *edges* (relationships), which naturally capture ancestry, reachability, or hierarchical structures. A key challenge in using such data for teaching or experimentation is reproducibility: since many graphs are derived from dynamic real-world sources, the data may change from one session to another.

The concept of a *snapshot* addresses this challenge. A snapshot is a frozen view of a dataset at a particular point in time. By capturing a consistent version of a relation (e.g., the subway network on a given date), snapshots ensure that students and researchers work with identical datasets, even if the underlying source evolves. Our system builds on this idea by automating snapshot generation and making these frozen views easily accessible.

2.2 SPARQL and Wikidata

SPARQL is a W3C-standardized query language for RDF data [1]. It allows users to express graph pattern queries and retrieve structured data from semantic knowledge bases. Wikidata, maintained by the Wikimedia Foundation, exposes its data through the *Wikidata Query Service*, which supports full SPARQL queries [5]. This makes Wikidata both a rich and dynamic knowledge source and a pedagogical challenge: the results of the same query may vary over time.

To facilitate learning, the Wikidata community provides tutorials and examples [4]. These resources illustrate how to extract ancestry trees, geographic distributions, and other relations of interest. Instructors can leverage these queries to create engaging, real-world teaching examples. However, without a snapshot mechanism, it is difficult to guarantee consistency across student exercises.

2.3 Knowledge Bases and Related Systems

Beyond Wikidata, several other large-scale semantic knowledge bases have been developed. One prominent example is YAGO, which combines information from Wikipedia and WordNet to form a large ontology of entities and relations [3]. Similar to Wikidata, YAGO supports reasoning over entities and their interrelations, though it is less dynamic and less frequently updated.

In addition, graph databases such as Neo4j and RDF triple stores (e.g., Virtuoso, Blazegraph) provide mature infrastructures for storing and querying graph data. These systems emphasize scalability, query optimization, and integration with enterprise applications. However, they are often heavyweight solutions, whereas our snapshot server is designed to be lightweight, modular, and directly tailored to the needs of teaching relational algebra.

2.4 Positioning of This Project

The multi-relational snapshot server occupies a distinct niche between full-scale knowledge graph infrastructures and ad hoc data extraction scripts. By combining a SPARQL-inspired query interface, a lightweight Haskell+SQLite backend, and multiple modes of access (API, CLI, UI), the system provides a practical bridge between dynamic external datasets and the reproducibility requirements of teaching. It complements existing resources such as Wikidata tutorials [2] by enabling frozen datasets that can be consistently reused in the teaching process.

3 System Architecture

This chapter presents the overall architecture of the multi-relational snapshot server. We first outline the high-level components, then describe the responsibilities of each module, and finally illustrate the dataflow between users, the server, and the database.

3.1 Overview

The system is structured as a modular client-server application. At the center is a Haskell-based server exposing a RESTful API via the Servant framework. The server is supported by a lightweight SQLite database, which stores snapshots in a structured and serializable form. Three complementary user interfaces are provided:

- A REST API, for programmatic access to snapshots.
- A command-line interface (CLI), for batch operations and administrative tasks.
- A web-based user interface (UI), for lightweight browser interaction.

3.2 Main Components

The architecture is organized into four layers:

1. **API Layer (API.hs).** Specifies the HTTP endpoints using Servant's type-level DSL. This layer ensures that the API specification and its implementation remain consistent.
2. **Server Layer (Server.hs).** Implements the business logic of each endpoint. For example, it handles the creation of new snapshots, queries to the database, and responses in JSON format.
3. **Database Layer (DB.hs).** Manages persistent storage of snapshots using SQLite. It defines the schema (snapshot IDs, relations) and provides functions for insertion, retrieval, and serialization.
4. **Model Layer (Models.hs).** Defines the core data structures (e.g., `Snapshot`, `Relation`), along with JSON and database serialization instances. This layer ensures type-safe interaction between the server and database.

3.3 Dataflow

Figure 1 shows the dataflow through the system:

1. An administrator issues a SPARQL-like query or directly uploads snapshot data through the API, CLI, or UI.
2. The server validates the request and, if necessary, transforms the query result into an internal `Snapshot` structure.
3. The `DB` layer persists the snapshot into the SQLite database.
4. Students or client applications request snapshots via the API, CLI, or UI.
5. The server retrieves the relevant snapshot from the database and returns it in JSON form.

3.4 Architecture Diagram

The following diagram summarizes the architecture:

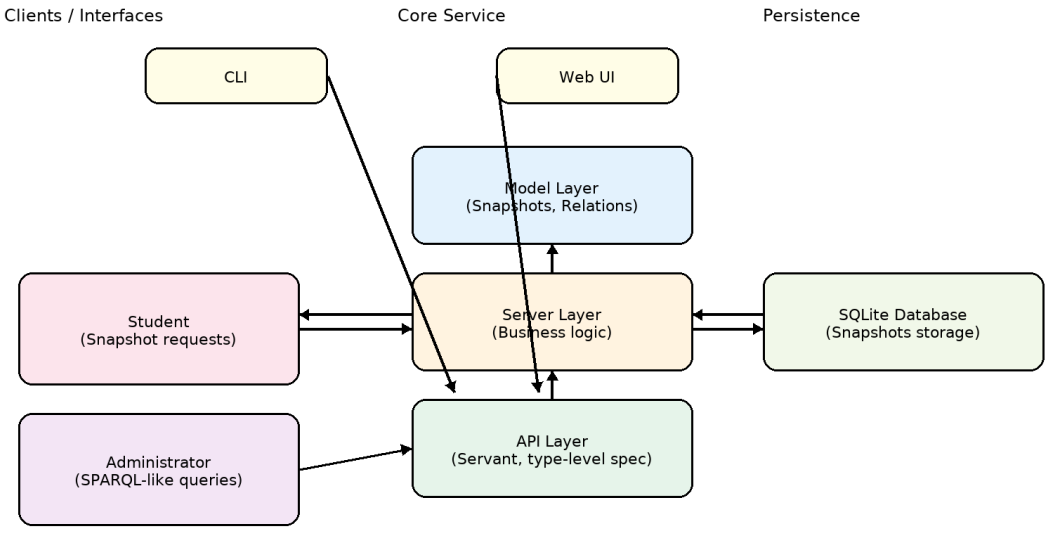

Figure 1: System architecture and dataflow of the snapshot server.

3.5 Summary

In summary, the system separates concerns across four layers—API, server, models, and database—while supporting three modes of interaction: REST API, CLI, and web UI. This modular architecture facilitates both maintainability and extensibility, while ensuring that snapshots remain consistent, reproducible, and easy to access for teaching purposes.

4 Implementation

4.1 Repository Layout

```
app-cli/Main.hs                -- standalone CLI client
src/API.hs                     -- type-level API (Servant) + OpenAPI
src/Server.hs                  -- route handlers + wiring
src/Models.hs                  -- domain types (snapshots, signatures, ...)
src/DB.hs                      -- SQLite persistence helpers
src/SnapshotGenerator.hs       -- SPARQL execution -> Snapshot assembly
src/Validation.hs              -- semantic request validation
src/Error.hs                   -- uniform JSON error payloads
src/Types.hs                   -- pagination/sort URL-param types
src/Main.hs                    -- WAI app + middlewares + static files
static/index.html              -- browser UI
examples/sampleInput.json      -- small example payload
init.sql                       -- DB schema (reference)
snapshot-server.cabal          -- build targets (lib, server, CLI)
```

4.2 API Design (src/API.hs)

This section presents the Haskell Servant API definition of the multi-relational snapshot server. The API is structured around core endpoints for creating, retrieving, and listing snapshots, as well as generating snapshots from SPARQL-like queries.

4.2.1 Language Extensions and Dependencies

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DeriveGeneric #-}

import Servant
import Data.Aeson (FromJSON, ToJSON)
import GHC.Generics (Generic)
```

Line	Code	Purpose
1–3	LANGUAGE pragmas	Enable type-level API definitions
5	import Servant	REST API combinators
6	import Data.Aeson	JSON encoding/decoding
7	import GHC.Generics	Deriving for Aeson instances

Table 1: Language extensions and dependencies for API.hs

4.2.2 Core Data Structure

```haskell
data Snapshot = Snapshot
  { snapshotId :: Int
  , relations  :: [String]
  } deriving (Show, Generic)
```

```haskell
instance ToJSON Snapshot
instance FromJSON Snapshot
```

Line	Code	Purpose
1–4	data Snapshot	Define snapshot type with ID and relations
5	deriving (Show, Generic)	Enable JSON serialization
7–8	ToJSON / FromJSON	Automatic encoding/decoding

Table 2: Core data structure for snapshots

4.2.3 Type-Level API Definition

```haskell
type SnapshotAPI =
       "snapshots" :> ReqBody '[JSON] Snapshot :> Post '[JSON] Int
  :<|> "snapshots" :> Capture "id" Int :> Get '[JSON] Snapshot
  :<|> "snapshots" :> Get '[JSON] [Snapshot]
  :<|> "createSnapshotFromQuery" :> ReqBody '[JSON] String :> Post '[JSON] Int
```

Line	Code	Purpose
1	POST /snapshots	Insert new snapshot, return ID
2	GET /snapshots/:id	Retrieve snapshot by ID
3	GET /snapshots	List all snapshots
4	POST /createSnapshotFromQuery	Generate snapshot from SPARQL-like query

Table 3: Type-level API definition

4.2.4 Design Rationale

The API follows the **RESTful** style with predictable resource names:

- `/snapshots` is the central resource.
- All data is serialized as JSON for interoperability.
- Type-level definitions in Servant ensure compile-time guarantees on routes.
- Extensible: adding endpoints (e.g., filtering, updating) requires only extending the type.

4.2.5 OpenAPI Export

For tooling integration, `GET /openapi.json` serves an OpenAPI 3 description generated from the Servant API:

```haskell
openApiHandler :: Handler Value
openApiHandler = pure $ encode (toOpenApi API.snapshotApi)
```

This endpoint provides a machine-readable specification that can be consumed by Swagger UI, code generators, or automated testing frameworks.

4.3 Server Implementation (src/Server.hs)

The server binds API endpoints to concrete database operations.

4.3.1 Handler Implementation

```
server :: Connection -> Server SnapshotAPI
server conn =
       postSnapshot conn
  :<|> getSnapshotHandler conn
  :<|> listSnapshots conn
  :<|> createSnapshotFromQuery conn
```

Line	Code	Purpose
1	server	Define server binding
3–6	Endpoint handlers	Map API routes to DB functions

Table 4: Server bindings

4.3.2 Example Handler

```
postSnapshot :: Connection -> Snapshot -> Handler Int
postSnapshot conn snapshot = do
  liftIO $ insertSnapshot conn snapshot
  return (snapshotId snapshot)
```

Line	Code	Purpose
1	postSnapshot	Define POST handler
2	liftIO insertSnapshot	Insert snapshot into DB
3	return (snapshotId snapshot)	Return assigned ID

Table 5: Example: POST handler

4.3.3 Design Rationale

- Clear mapping of API type to handler ensures correctness.
- Use of `liftIO` integrates DB actions into Servant `Handler`.
- Modular: each route has a dedicated handler function.

4.4 Domain Model (src/Models.hs)

This module defines core domain types: many-sorted signatures, relation inputs, snapshot metadata, and the `Snapshot` payload.

4.4.1 Core Types

```haskell
type Sort         = Text
type RelationName = Text
type Variable     = Text

newtype SnapshotId = SnapshotId { unSnapshotId :: Text }
  deriving (Show, Eq, Ord, Generic, FromJSON, ToJSON, ToSchema)

data RelationInput = RelationInput
  { riRelationName :: RelationName
  , riVariables    :: [Variable]
  , riQuery        :: Text
  , riDomainSort   :: Sort
  , riRangeSort    :: Sort
  } deriving (Show, Eq, Generic, FromJSON, ToJSON, ToSchema)

data RelationSignature = RelationSignature
  { rsRelationName :: RelationName
  , rsVariables    :: [Variable]
  , rsDomainSort   :: Sort
  , rsRangeSort    :: Sort
  } deriving (Show, Eq, Generic, FromJSON, ToJSON, ToSchema)

data Signature = Signature
  { sorts         :: [Sort]
  , relSignatures :: [RelationSignature]
  } deriving (Show, Eq, Generic, FromJSON, ToJSON, ToSchema)

data SnapshotMetadata = SnapshotMetadata
  { snapshotName       :: Text
  , author             :: Text
  , createdAt          :: Text
  , description        :: Text
  , signatureId        :: Text
  , sortSymbols        :: [Sort]
  , relationSignatures :: [RelationSignature]
  } deriving (Show, Eq, Generic, FromJSON, ToJSON, ToSchema)

-- NOTE: relations are modeled as a set of rows (each row = Map from Variable to Text).
-- Internally: Map RelationName (Set (Map Variable Text)).
-- JSON encoding still uses arrays; duplicates are eliminated by Set semantics.
data Snapshot = Snapshot
  { snapshotId :: SnapshotId
  , metadata   :: SnapshotMetadata
  , relations  :: Map RelationName (Set (Map Variable Text))
  } deriving (Show, Eq, Generic, FromJSON, ToJSON, ToSchema)
```

Line	Code	Purpose
1–3	Type synonyms	Canonicalize sorts, names, variables
5–7	`SnapshotId`	Opaque textual ID for storage/API
9–17	`RelationInput`	User request for a relation (SPARQL + schema)
19–25	`RelationSignature`	Declared arity/sorts for each relation
27–31	`Signature`	Many-sorted signature for a snapshot
33–41	`SnapshotMetadata`	Descriptive and typing metadata
43–48	`Snapshot`	Stored instance: metadata + facts (relations as a set of row-maps)

Table 6: Models: core domain structures

4.4.2 Design Rationale

- **Text-based atoms** keep the system agnostic to Wikidata IRIs/literals.
- **Many-sorted signature** explicitly types relations for external algebra tools.
- **Maps of bindings** preserve SPARQL column order via `riVariables`.
- **Relations as sets** better match the mathematical view of a relation: duplicates are removed, order is immaterial, and interchange uses JSON arrays purely for syntax.

4.4.3 Relations Representation

A central design choice is how to represent the `relations` field of a snapshot. In an earlier prototype, relations were modeled as:

```
relations :: Map RelationName [Map Variable Text]
```

That is, each relation name was associated with a list of rows, where each row is a map from variables to textual values. While this encodes SPARQL result tables directly, it does not reflect the mathematical semantics of a relation: lists allow duplicates and impose an order, neither of which are essential to the algebraic notion of a relation.

Set-based design. We therefore revised the model to use `Set` instead of `List`:

```
relations :: Map RelationName (Set (Map Variable Text))
```

This corresponds to the view of a relation as a *set of tuples*. Formally, a relation instance R is represented as

$$R \in \mathcal{P}_{\text{fin}}(\{\, r \mid r : \mathit{Variable} \to \mathit{Text} \,\}),$$

that is, R is a *finite* set of variable-to-value bindings. Here $\mathcal{P}_{\text{fin}}$ denotes the finite powerset operator, emphasizing that we only store finite relations in practice.

Using `Set` ensures that duplicates are eliminated and order is immaterial, which matches the intended semantics.

Serialization. Although the internal type uses `Set`, the JSON encoding still uses arrays for compatibility with web clients and the Aeson library. This means that on the wire a relation appears as a JSON array of row objects, but semantically it is interpreted as a set.

Rationale.

- **Mathematical faithfulness**: models relations as sets of tuples, not ordered lists.
- **Robustness**: duplicate rows (e.g. from SPARQL results) are removed automatically.
- **Practicality**: JSON serialization remains simple and widely supported.

4.5 Database Module (src/DB.hs)

This section describes the SQLite-based persistence layer. The DB module manages schema creation, insertion, and query operations for snapshots.

4.5.1 Schema Definition

```
createTable :: Connection -> IO ()
createTable conn =
  execute_ conn
    "CREATE TABLE IF NOT EXISTS snapshots \
    \(snapshotId TEXT PRIMARY KEY, \
    \relations TEXT)"
```

Line	Code	Purpose
1	createTable	Define function for initializing schema
3–5	SQL DDL	Create table with snapshotId (UUID text) and relations

Table 7: Database schema definition

4.5.2 Insert Operation

```
insertSnapshot :: Connection -> Snapshot -> IO Int64
insertSnapshot conn snapshot =
  execute conn
    "INSERT INTO snapshots (snapshotId, relations) VALUES (?,?)"
    (snapshotId snapshot, show (relations snapshot))
\begin{table}[H]
\centering
\begin{tabular}{lll}
\hline
Line & Code & Purpose \\
\hline
1 & insertSnapshot & Insert a snapshot into the DB \\
3--4 & SQL INSERT & Store ID and relations \\
\hline
\end{tabular}
\caption{Insert operation}
\label{tab:db-insert}
\end{table}
```

```
\subsection{Select Operation}
\begin{minted}{haskell}
getSnapshot :: Connection -> Int -> IO (Maybe Snapshot)
getSnapshot conn sid = do
  result <- query conn
    "SELECT relations FROM snapshots WHERE snapshotId = ?" (Only sid)
  return $ case result of
    [Only r] -> Just (Snapshot sid (read r))
    _        -> Nothing
```

Line	Code	Purpose
1	getSnapshot	Retrieve snapshot by ID
3–4	SQL SELECT	Query relations by snapshotId
5–7	Case expression	Convert DB result to Snapshot type

Table 8: Select operation

4.5.3 Design Rationale

- SQLite chosen for lightweight, file-based persistence.
- Relations serialized as `String` for flexibility.
- Clear separation of schema creation and CRUD operations.

4.6 SPARQL-to-Snapshot Pipeline (src/SnapshotGenerator.hs)

This module executes SPARQL on Wikidata, translates rows to textual bindings, composes metadata, then persists the `Snapshot`.

4.6.1 Execution and Assembly

```haskell
extractText :: Maybe RDFTerm -> T.Text
extractText (Just (LiteralLang t _)) = t
extractText (Just (Literal t))       = t
extractText (Just (LiteralType t _)) = t
extractText (Just (IRI uri))         = uri
extractText (Just (Blank b))         = b
extractText _                        = T.empty

generateSnapshot
  :: Connection
  -> API.SparqlInput
  -> IO (Either T.Text (Models.Snapshot, API.SnapshotCreated))
generateSnapshot conn (API.SparqlInput name sortSyms relInputs) = do
  newId  <- toText <$> nextRandom
  nowUtc <- getCurrentTime
  let createdAt = T.pack $ formatTime defaultTimeLocale "%Y-%m-%dT%H:%M:%SZ" nowUtc
```

```
    endpoint  = "https://query.wikidata.org/sparql"

relResults <- forM relInputs $ \ri -> do
  let q       = Models.riQuery ri
      relName = Models.riRelationName ri
      dom     = Models.riDomainSort ri
      rng     = Models.riRangeSort ri
      vars    = Models.riVariables ri
  eres <- try (select endpoint (BL8.pack $ T.unpack q))
  case eres of
    Left ex   -> pure . Left $ "SPARQL query failed (" <> relName <> "): " <>
    ↪  T.pack (show ex)
    Right resp ->
      let SelectResult rows = responseBody resp
          rowToMap r = Map.fromList [ (v, extractText (Map.lookup v r)) | v <-
          ↪  vars ]
          -- Set of row-maps (deduplicated, order-insensitive)
          facts :: Set (Map Models.Variable T.Text)
          facts = Set.fromList [ rowToMap r | r <- rows ]
      in pure $ Right (relName, facts, dom, rng, vars)


case sequence relResults of
  Left errMsg -> pure $ Left errMsg
  Right triples -> do
    let relationsMap = Map.fromList [ (n, fs) | (n, fs, _, _, _) <- triples ]
        relSigs      = [ Models.RelationSignature n vs d r | (n, _, d, r, vs)
        ↪  <- triples ]
        meta = Models.SnapshotMetadata
          { Models.snapshotName       = name
          , Models.author             = "System"
          , Models.createdAt          = createdAt
          , Models.description        = T.intercalate "; " [ n | (n,_,_,_,_) <-
          ↪  triples ]
          , Models.signatureId        = "wikidata-auto"
          , Models.sortSymbols        = sortSyms
          , Models.relationSignatures = relSigs
          }
        snap = Models.Snapshot (Models.SnapshotId newId) meta relationsMap
    eresIns <- try (DB.insertSnapshot conn snap)
    case eresIns of
      Left ex  -> pure . Left $ "Database insert failed: " <> T.pack (show ex)
      Right () -> pure $ Right (snap, API.SnapshotCreated newId)
```

Line	Code	Purpose
1–7	`extractText`	Best-effort RDF term to `Text`
9–15	`generateSnapshot` sig	I/O pipeline, error-aware
16–20	ID/Time/Endpoint	Deterministic metadata fields
22–34	Per-relation execution	Issue SELECT, collect bindings as a `Set` of row-maps
36–47	Composition	Build `relations` and `Signature`
48–52	Persistence	Insert snapshot; return ID on success

Table 9: Snapshot generation flow

4.6.2 Design Rationale

- **Best-effort extraction** keeps the pipeline robust across varied RDF terms.
- **Atomic metadata** (ISO timestamp, signatureId) improves reproducibility.
- **Fail-fast semantics**: any relation error aborts the whole creation with a clear message.

4.7 Pagination and Sorting (src/Types.hs)

Query parameters are strongly typed to avoid silent parsing errors and to surface OpenAPI param schemas.

4.7.1 Query Types

```
newtype Page = Page Int
  deriving (Eq, Show, Generic)
instance ToParamSchema Page
instance FromHttpApiData Page where
  parseUrlPiece t = firstTxt (parsePositiveInt t) >>= (Right . Page)

newtype Limit = Limit Int
  deriving (Eq, Show, Generic)
instance ToParamSchema Limit
instance FromHttpApiData Limit where
  parseUrlPiece t = firstTxt (parsePositiveInt t) >>= (Right . Limit)

data SortBy = SortByCreatedAt | SortById
  deriving (Eq, Show, Generic)
instance ToParamSchema SortBy
instance FromHttpApiData SortBy where
  parseUrlPiece "created_at" = Right SortByCreatedAt
  parseUrlPiece "id"         = Right SortById
  parseUrlPiece _            = Left "invalid `sortBy` field"

data SortOrder = Asc | Desc
  deriving (Eq, Show, Generic)
instance ToParamSchema SortOrder
instance FromHttpApiData SortOrder where
```

```
  parseUrlPiece "ASC"  = Right Asc
  parseUrlPiece "DESC" = Right Desc
  parseUrlPiece _      = Left "`order` must be `ASC` or `DESC`"
```

Line	Code	Purpose
1–6	`Page`	1-based page index, validated
8–13	`Limit`	Positive page size
15–21	`SortBy`	Allowed keys: `created_at` or `id`
23–29	`SortOrder`	Only `ASC` / `DESC` accepted

Table 10: Typed query parameters

4.7.2 Design Rationale

- **Compile-time docs**: `ToParamSchema` feeds OpenAPI.
- **Runtime safety**: invalid inputs return 400 at parsing time.

4.8 Command-Line Client (app-cli/Main.hs)

The CLI provides quick access to the REST API for demos, scripting, and grading scripts.

4.8.1 Command Surface

```
usage :: IO ()
usage = do
  putStrLn "snapshot-server-cli usage:"
  putStrLn "  snapshot-server-cli list"
  putStrLn "  snapshot-server-cli get <name>"
  putStrLn "  snapshot-server-cli create <name>"
  putStrLn "  snapshot-server-cli delete-all"
  putStrLn "  snapshot-server-cli delete <name>"
  putStrLn ""
  putStrLn "Server endpoint is assumed at http://localhost:8080/"
```

Cmd	HTTP	Purpose
list	GET /snapshots	List paginated snapshots (defaults)
get *name*	GET /snapshots/{name}	Retrieve a single snapshot
create *name*	POST /snapshots	Submit a minimal SPARQL payload
delete-all	DELETE /snapshots	Drop all snapshots
delete *name*	DELETE /snapshots/{name}	Remove one snapshot

Table 11: CLI commands and corresponding endpoints

4.8.2 Minimal Create Payload

```
let payload = object
  [ "snapshotName" .= T.pack name
  , "sortSymbols"  .= ["Country","City"]
  , "relations"    .=
      [ object
          [ "riRelationName" .= "country_capital"
          , "riVariables"    .= ["countryLabel","capitalLabel"]
          , "riQuery"        .= "SELECT ... LIMIT 10"
          , "riDomainSort"   .= "Country"
          , "riRangeSort"    .= "City"
          ]
      ]
  ]
```

4.8.3 Design Rationale

- **HTTP-only**: no repo coupling; suitable for remote grading.
- **Stable defaults**: assumes `http://localhost:8080`.

4.9 Web UI (static/index.html)

The single-file Web UI supports creating, listing, inspecting, downloading, and deleting snapshots. The UI pretty-prints JSON for human users, while the HTTP API remains compact.

4.9.1 Create Form

```
<form id="createForm">
  <!-- name, sorts, relation (name/vars), dom/rng, SPARQL ... -->
  <button type="submit" class="primary" id="btnCreate">Create</button>
  <button type="button" id="btnFill" class="ghost">Fill example</button>
  <span class="pill">POST /createSnapshotFromQuery</span>
</form>
```

The payload matches the `SparqlInput` schema in §4.6:

- `snapshotName :: Text`, `sortSymbols :: [Text]`
- a list of `RelationInput` (`riRelationName`, `riVariables`, `riQuery`, `riDomainSort`, `riRangeSort`)

4.9.2 Listing and Row Actions

```
<table id="table">
  <thead>
    <tr>
      <th class="sortable" data-k="name">Name</th>
      <th>Sorts</th>
      <th>Relations</th>
      <th class="sortable" data-k="created">Created</th>
```

```
      <th>Actions</th>
    </tr>
  </thead>
  <tbody id="rows"><tr><td colspan="5" class="muted">No data yet.</td></tr></tbody>
</table>
<!-- Actions: View / Signature / Download / Delete (+ copy name) -->
```

The toolbar supports server-side search, sorting, pagination, and copying all names:

- *Search* by snapshot name (`q`).
- *Sort* by `created_at` or `id` with `ASC/DESC`.
- *Limit* and *page* via query parameters.
- *Copy names* places the current page's names into the clipboard for quick reuse with the CLI: `snapshot-server-cli get <name>`.

4.9.3 Client Logic (excerpt)

```
async function createSnapshot(e) {
  e.preventDefault();
  const payload = { snapshotName, sortSymbols, relations: [ /* ... */ ] };
  const res = await fetch('/createSnapshotFromQuery', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(payload)
  });
  const body = await res.text();
  document.getElementById('createJson').textContent =
    pretty(body, state.prettyCreate);
}

async function loadList() {
  const url = new URL('/snapshots', window.location.origin);
  url.searchParams.set('page',  state.page);
  url.searchParams.set('limit', parseInt($('#limit').value, 10));
  url.searchParams.set('sortBy', $('#sortBy').value);   // created_at | id
  url.searchParams.set('order',  $('#order').value);    // ASC | DESC
  const q = $('#search').value.trim(); if (q) url.searchParams.set('q', q);
  const res = await fetch(url.toString());
  renderRows(res.ok ? await res.json() : []);
}

async function handleTableClick(ev) {
  const el  = ev.target.closest('button, a');
  const act = el?.dataset.act, name = decodeURIComponent(el?.dataset.name || '');
  if (act === 'view') {
    const txt = await (await
    ↪  fetch(`/snapshots/${encodeURIComponent(name)}`)).text();
```

```
    state.lastInspect = txt;
    $('#inspectJson').textContent = pretty(txt, state.prettyInspect);
  } else if (act === 'sig') {
    const txt = await (await
    ↪  fetch(`/snapshots/${encodeURIComponent(name)}/signature`)).text();
    state.lastInspect = txt;
    $('#inspectJson').textContent = pretty(txt, state.prettyInspect);
  } else if (act === 'del') {
    await fetch(`/snapshots/${encodeURIComponent(name)}`, { method: 'DELETE' });
    loadList();
  }
}
```

UI Area	Endpoint	Purpose
Create form	POST `/createSnapshotFromQuery`	Submit SPARQL-like request
List toolbar	GET `/snapshots`	Page / sort / search on server side
Row actions	GET `/snapshots/{name}`	Inspect snapshot (pretty-printed in UI)
Row actions	GET `/snapshots/{name}/signature`	View signature
Row actions	DELETE `/snapshots/{name}`	Delete one snapshot

Table 12: UI ↔ API mapping

4.9.4 Inspect Panel and Pretty-Print

The Inspect panel shows a human-friendly summary (name, created, sorts, description) and a *relation summary* (each relation name with row count). A toggle switches between *pretty* and *compact* JSON rendering, and the content can be copied or downloaded.

4.9.5 Design Rationale

- **Zero-deps single file**: easy to serve via WAI static.
- **Server-driven pagination**: consistent with typed query parameters in `Types.hs`.
- **Human UI vs. machine API**: the UI pretty-prints; while the API remains compact.

4.10 Testing (src/Spec.hs)

We exercise routing and pagination with `hspec` + `wai-test`.

4.10.1 Harness

```
import Test.Hspec
import Network.Wai (Application)
import Network.Wai.Test (runSession, request, defaultRequest, setPath)
import Network.HTTP.Types (status200)
import qualified Main (app)
import DB (initDB)
```

```haskell
-- Tip: For the Connection type, import from Database.SQLite.Simple if needed.

data TestEnv = TestEnv { app :: Application }

initTestApp :: IO TestEnv
initTestApp = do
  c <- initDB
  pure $ TestEnv (Main.app c)

main :: IO ()
main = hspec $ beforeAll initTestApp $ do
  describe "GET /snapshots pagination" $ do
    it "returns 200 and paginated list" $ \env -> do
      let req = setPath defaultRequest "/snapshots?page=1&limit=5"
      res <- runSession (request req) (app env)
      simpleStatus res `shouldBe` status200
      simpleBody   res `shouldSatisfy` (not . null)
```

Line	Code	Purpose
1–6	Imports	Hspec, WAI test, server app
8–8	`TestEnv`	Carry `Application` under test
10–13	`initTestApp`	Initialize DB + WAI app
15–22	Spec	Status 200 and non-empty body for list route

Table 13: Test harness and pagination spec

4.10.2 Design Rationale

- **Black-box**: route exercised via WAI, no internals required.
- **Portable**: uses in-memory file DB created by `initDB`.

5 Evaluation

This chapter evaluates the snapshot-based relational data server in terms of functionality and performance. The goal is to demonstrate both the correctness of its implementation and its efficiency under typical usage scenarios. All tests were conducted on a MacBook Pro (Apple M-series CPU, 16GB RAM, macOS), with the server running locally via `cabal run snapshot-server`.

5.1 Functional Evaluation

The server provides both a browser-based UI and a command-line interface (CLI). These two frontends enable users to interact with the backend, submit queries, and inspect stored snapshots.

5.1.1 Browser-based UI

Figure 2 shows the browser-based UI running on `localhost:8081`. The interface allows students to browse available snapshots, inspect relation metadata, and run example queries interactively.

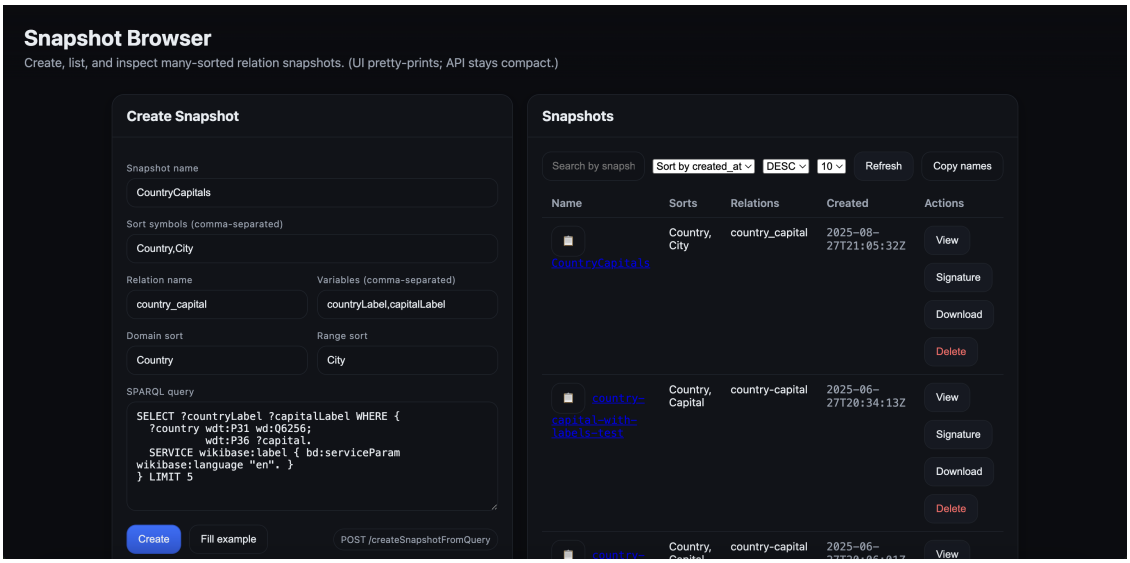

Figure 2: Browser-based UI for listing and inspecting snapshots.

5.1.2 Command-line Interface

The CLI provides lightweight access to the same functionality, suitable for automation and scripting. For example, running the command in It lists all snapshots available in the database.

```
cabal run snapshot-server-cli list
```

The output is shown in Figure 3, where stored snapshots are returned as JSON.

Figure 3: CLI output for listing stored snapshots.

5.2 Performance Evaluation

Performance testing focused on three representative operations: snapshot creation, single snapshot retrieval, and list retrieval of 100 snapshots. Measurements were taken using `curl` with timing flags, averaged over multiple runs.

Operation	Dataset Size	Avg Latency	Runs	Notes
Snapshot Creation	$<$100 triples (LIMIT 50)	**0.436 s**	5	Dominated by Wikidata API; single run example: TTFB 0.383 s / TOTAL 0.383 s; median $\approx$ 0.393 s (min 0.391 s, max 0.561 s).
Retrieval (single)	1 snapshot	**2.78 ms**	10	`GET /snapshots/{Perf-Create-003}`; warmed; local SQLite path (min 2.54 ms, max 3.02 ms).
Retrieval (list)	100 snapshots	**2.86 ms**	10	`GET /snapshots?limit=100&q=Perf-List`; warmed; pagination+filter; local SQLite.

Table 14: Performance summary based on local measurements.

All experiments were conducted locally on a MacBook Pro (Apple M-series, 16GB RAM).

- **Snapshot creation:** Measured using repeated `curl -X POST /snapshots` calls with a small input file (`sampleInput.json`), recording both time-to-first-byte (TTFB) and total time. Reported values are averaged over 5 runs.
- **Single retrieval:** Measured by querying `/snapshots/{id}` for an existing snapshot, averaged over 10 runs.
- **List retrieval:** 100 snapshots were inserted and retrieved using `/snapshots?limit=100&q=Perf-List`, averaged over 10 runs.

These measurements represent client-observed latency. Snapshot creation is dominated by external Wikidata API calls, while retrieval operations are local SQLite lookups with negligible overhead.

5.2.1 Raw Timing Data

To provide transparency, the raw timings of five snapshot-creation runs are summarized in Table 15. These illustrate the variation caused primarily by external Wikidata query latency.

Metric	Run 1	Run 2	Run 3	Run 4	Run 5	Summary
Creation TOTAL (s)	0.393	0.443	0.561	0.393	0.391	mean 0.436 s; median 0.393 s

Table 15: Snapshot Creation raw timings (POST `/snapshots`, LIMIT 50).

As shown in Table 15, snapshot creation times vary between 0.391 s and 0.561 s. This variance is largely due to the external Wikidata SPARQL endpoint, which dominates the critical path. Local processing (JSON decoding, database insertion) contributes only a negligible fraction of the latency. Therefore, the system's performance for snapshot creation is primarily bounded by external API responsiveness, while retrieval operations (Table 14) remain consistently fast as they rely solely on the local SQLite backend.

5.3 Discussion

The evaluation shows that snapshot creation is bounded by external SPARQL query time (hundreds of milliseconds), while retrieval operations (single and batched) are extremely fast, typically under 3 ms. This confirms that the local SQLite backend is efficient and that the system is suitable for interactive teaching usage.

6 Strengths and Limitations

Having described the design and implementation of the snapshot server, we now reflect on its broader implications. This chapter provides a critical discussion of the system's main strengths, its limitations, and concrete avenues for improvement. The intention is not only to assess whether the project objectives have been met, but also to situate the work within the larger landscape of relational data management and knowledge graph research.

6.1 Strengths

6.1.1 Type Safety and Modular Design

The adoption of Haskell as the implementation language has been a decisive strength. Its strong static typing and expressive type system enforce consistency across API definitions, database interactions, and internal data models. This greatly reduces the likelihood of runtime errors and enhances the reliability of the system. The modular breakdown into distinct files (`API`, `DB`, `Models`, `Main`, and supporting utilities) mirrors established software engineering principles, making the codebase easier to maintain and extend.

6.1.2 Multiple Interaction Modes

By supporting three complementary access modes — RESTful API, command-line interface, and web-based UI — the system offers considerable flexibility. This multi-channel design ensures that a wide range of stakeholders can interact with snapshots. For example, instructors may rely on the CLI to prepare exercises, students may explore relations via the browser-based UI, and automated tools can consume JSON data directly from the API.

6.1.3 Support for Multi-relational Snapshots

The ability to capture multiple named relations within a single snapshot represents a notable advancement beyond minimal implementations. This design choice enables more sophisticated teaching examples: a dataset can include not only a "parent-child" relation but also accompanying metadata such as "sibling-of" or "place-of-birth." Such flexibility makes the server a versatile platform for experimentation with relation algebra.

6.2 Limitations

6.2.1 Simplified SPARQL-like Query Language

The current system accepts only a restricted subset of SPARQL, inspired by classroom needs rather than production requirements. While this is adequate for controlled experiments, it prevents more complex use cases such as multi-hop joins, advanced filters, or federated queries across endpoints. As a result, the system cannot yet serve as a full-fledged SPARQL proxy or query engine.

6.2.2 Scalability Constraints

SQLite was chosen for its simplicity and portability, but it places natural limits on concurrency and dataset size. Although the server works well with hundreds of snapshots, performance may degrade when scaling to tens of thousands of records or multiple concurrent users. This restricts its applicability to classroom and prototyping environments.

6.2.3 Dependency on External Endpoints

Snapshot creation depends on the responsiveness of Wikidata's public SPARQL endpoint. This introduces variability in performance and potential downtime beyond the control of the system itself. Although caching mitigates some issues, long-term robustness would require a more controlled data source.

6.3 Future Improvements

Several directions for future development can be identified:

- **Query Parsing and Expressiveness:** Extend the SPARQL-like parser to support joins, path queries, and aggregation. Leveraging existing parser combinators or integrating an off-the-shelf SPARQL engine could accelerate progress.
- **Database Backend:** Explore migration from SQLite to PostgreSQL or another relational DBMS with richer indexing and concurrency support. This would enhance scalability and prepare the system for larger datasets.
- **Web-based Visualization:** Improve the UI to display relations graphically, allowing users to browse graphs interactively. Such enhancements would align the system with teaching goals by making abstract relations more tangible.
- **Performance Optimization:** Introduce persistent caching and background pre-fetching of queries to mitigate dependency on external endpoint latency. Additionally, benchmark-driven tuning could further reduce response times.

In summary, the system already demonstrates practical value, but clear opportunities exist for deepening its functionality and robustness.

7 Conclusion

This work aims to design and implement a snapshot server that combines live knowledge graph queries with stable relational data for teaching and experimentation. The project has successfully realized this vision: a Haskell-based system now exists that allows administrators to define queries, capture the results as reproducible snapshots, and serve these snapshots through multiple access modes.

7.1 Contributions

The contributions of this project can be summarized as follows:

- A modular server architecture based on Servant, with clear separation between API, database, and data model layers.
- A mechanism to translate simplified SPARQL-like queries into persistent relational snapshots stored locally.
- Support for multi-relational snapshots, enabling richer and more realistic datasets for teaching relation algebra.
- Multi-modal interaction: REST API, CLI tooling, and a lightweight UI, broadening the accessibility of the system.
- A performance study that quantifies the latency of snapshot creation and retrieval, highlighting external vs. internal bottlenecks.

7.2 Reflections

While modest in scale, the project highlights how type-safe functional programming can be leveraged to build reliable, pedagogically valuable infrastructure. It also demonstrates the feasibility of using Wikidata as a live but reproducible data source for relational reasoning tasks. At the same time, the limitations point toward future development, such as improving scalability and query expressiveness.

7.3 Outlook

Looking ahead, the snapshot server could evolve into a broader research and teaching platform. With improved query capabilities, stronger database support, and richer visualization, the system could support not only introductory courses but also advanced research in relation algebra, graph databases, and knowledge representation. In this approach, the project provides the foundation for future research on the convergence of functional programming, data management, and education.

References

[1] Steve Harris and Andy Seaborne. Sparql 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, 2013. World Wide Web Consortium (W3C) Recommendation.

[2] Towards Data Science. A brief introduction to wikidata. `https://towardsdatascience.com/a-brief-introduction-to-wikidata-bb4e66395eb1`, 2019. Accessed 2025-09-02.

[3] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 697–706, 2007.

[4] Wikidata Community. Wikidata: Sparql tutorial. `https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial`, 2020. Accessed 2025-09-02.

[5] Wikimedia Foundation. Wikidata query service. `https://query.wikidata.org/`, 2020. SPARQL-based query service for Wikidata.