Title

SRFI XYZ: Generic Tuple Store Database

Author

Amirouche Boubekki

Status

???

Abstract

This library is a generic approach to the database abstractions known as triple-store and quadstore. Generic Tuple Store Database implements n-tuple ordered sets and associated primitives for working with them in the context of data management.

Rationale

The industry standard for durable data storage namely Relational Database Management Systems (RDBMS) do not blend nicely into Scheme. In particular, the SQL programming language is very difficult to embed in Scheme without fallbacks to string interpolations. This could explain the poor support of RDBMS in Scheme implementations.

This SRFI propose a powerful database abstraction for storing and querying data that integrates well in existing Scheme landscape by re-using SRFI-9, SRFI-128, SRFI-146, SRFI-158 and it relies on Ordered Key Value Store SRFI. This SRFI comes with an memory storage engine that support transactions. It can be adapted to easily implement durability relying on one of the various existing ordered key-value store libraries available in the wild.

This SRFI does not overlap with existing SRFIs.

Specification

This specification defines four disjoint types:

- `nstore` is a n-tuple store database where n is fixed.
- `engine` is exposes a lexicographically ordered key store interface as described in Ordered Key Value Store SRFI.
- `transaction` is a handle over a running transaction.
- `variable` is an immutable object associcated with a symbol.

Also an implementation must rely on srfi-158 generators to implement streams and srfi-146 mapping hash to implement bindings.

The first section describe the public interface of the tuple store abstraction. The second describe the `engine` type that allows to make the storage mechanic configurable.

Pluggable Storage Engine

Storage engine type expose the interface described in Ordered Key Value Store SRFI so that it possible to swap one storage engine with another seamlessly.

`(engine database close begin! commit! rollback! ref set! rm! range)`

Returns an object suitable to pass to `make` procedure.

Generic Tuple Store Database

The database abstraction that is described in this section is an ordered set of tuples with n objects. A given store will always contain tuples of the same length. Tuples are always passed as rest arguments to the database procedures. Because of that they might be represented as lists in the application using the store.

`(make engine . items)`

Returns an object that can be passed as first argument to the procedures "decorated" with `transactional`. `ENGINE` is described above. `ITEMS` describe the names given to a tuple's items. It should be a list of symbols.

In the following `ITEMS` is always a list of scheme objects. What is accepted in `ITEMS` list depends on the implementation.

`(close nstore)`

Close `NSTORE`.

`(transactional proc) -> procedure`

Takes a procedure as argument that takes a transaction object as first argument and zero or more arguments. `transactional` will return a procedure that can be called with a transaction object or a store object as first argument. `transactional` has the responsability to begin, commit, rollback and retry transaction when appropriate. In case of exception during the transaction, it is up to `transactional` to rollback the transaction and re-raise the exception if appropriate. In the case where the implementation implements transaction retry, `PROC` must be idempotent.

`transactional` allows to compose procedures that interact with the database without manually having to manage transactions.

A procedure is said to be decorated with `transactional` when it is passed to `transactional` and assigned to a scheme variable:

```
(define everything (transactional %everything))
```

In the above `%everything` is decorated by `transactional` and assigned to `everything`.

`(everything some)`

Returns the items that are in `NSTORE` returned as an SRFI-158 generator of lists. This is a procedure for debugging. Actual querying of the database must be done with `from` and `where` procedures. The stream must be ordered as described in Ordered Key Value Store SRFI.

`(ask? some . items)`

Must be decorated with transactional so that it is possible to pass a transaction or a nstore object.

Returns `#t` if `ITEMS` is present in the store associated with `TRANSACTION`. Otherwise it returns `#f`.

`(add! some . items)`

Must be decorated with transactional so that it is possible to pass a transaction or a nstore object as first argument.

Add `ITEMS` to the store associated with `TRANSACTION`. If `ITEMS` is already in the associated store it does nothing. Returned value is unspecified.

`(rm! some . items)`

Must be decorated with transactional so that it is possible to pass a transaction or a nstore object as first argument.

Removes `ITEMS` from the store associated with `TRANSACTION`. It does nothing if `ITEMS` is not in the store. Returned value is unspecified.

`(var name)`

Returns an object of a disjoint type `variable` associated with the symbol `NAME`.

`(var? obj) → boolean`

Returns `#t` if `OBJ` is a variable as returned by `var` procedure. Otherwise it returns `#f`.

`(var-name variable) → symbol`

Returns the symbol name associated with `VARIABLE`. If `VARIABLE` is not a variable is the sense of `var?` the returned value is unspecified.

`(from some . pattern) → generator`

Must be decorated with transactional so that it is possible to pass a transaction or a nstore object as first argument.

Returns a generator of bindings where variables in sens of `var?` of `PATTERN` are bound against one or more *matching* tuples from the store associated with `TRANSACTION`. The implementation must return a srfi-158 generator of srfi-146 hash mappings.

The returned generator is called seed generator because it doesn't rely on an existing generator of bindings.

Note: Making the stream ordered, in this case, might not be worthwhile. If there is an interest it might be the subject of a future srfi. Also it will be required to define an efficient mapping that remembers key-value insertion order which will allow to define a total order between bindings.

`(where some . pattern) → procedure → generator`

Must be decorated with transactional so that it is possible to pass a transaction or a nstore object as first argument.

Returns a procedure that takes a generator of bindings as argument and returns a generator of bindings where variables of `PATTERN` are bound to one or more *matching* tuples from the store associated with `TRANSACTION`.

Here is an example use of `where` in conjunction with `from` that fetch the title of article that have scheme as tag:

```
(define seed (from transaction 'article (var 'uid) 'tag 'scheme))
(define articles ((where transaction 'article (var 'uid) 'title (var 'title)) seed))
```

`articles` variable contains a generator of bindings with `uid` and `title` keys of all the articles that have scheme as tag.

Note: The generator returned by `where` is flat. It is NOT a generator of generators.

Implementation

wip

Acknowledgements

Credits goes first to Cognitect for creating Datomic database which inspired this work. StackOverflow user zhoraster helped pin the mathematics behind the generic implementation of n-tuple stores and Mike Earnest provided an algorithm to compute the minimal set of tuple items permutations that allows efficient querying. Also, some credits goes FoundationDB for the lexicographic packing algorithms and the idea of the transactional procedure. The author would like to thank Arthur A. Gleckler, Marc Nieper-Wißkirchen for getting together SRFI 146 and Shiro Kawai, John Cowan, Thomas Gilray for working on SRFI 158.

Copyright