Lobos

Introduction

Presentation given to the Bonjure group on 2011/01/21.

Outline

- Overview & basic concepts
- **Past**: history and development highlights
- **Present**: examples, project structure and architecture
- **Future**: migrations and declarative schema manipulation

Overview

Clojure code $\longrightarrow$ SQL Statement(s)*

```
(create
 (table :users
   (integer :id :auto-inc
                :primary-key)
   (varchar :name 100 :unique)
   (check :min_length
          (> :length/name 1))))
```

```
CREATE TABLE "users" (
  "name" VARCHAR(100),
  "id" SERIAL,
  CONSTRAINT "min_length"
    CHECK (length("name") > 1),
  CONSTRAINT "users_unique_name"
    UNIQUE ("name"),
  CONSTRAINT "users_primary_key_id"
    PRIMARY KEY ("id"))
```

Note that the syntax is partially inspired by `ActiveRecord::Migration`

*This example use the PostgreSQL backend.

Basic Concepts

- **Abstract Schema Definition**: Used internally to represent a database schema.
- **Action**: Functions executing imperative SQL statements.
- **AST**: Intermediate database-agnostic representation of SQL statements.

Past

History

- Lobos is based on the original ClojureQL project which was based on SchemeQL.
- It's a complete rewrite of the DDL part.
- Lobos is less than a month old and a one-man effort for now.

Development Highlights

- Uses most Clojure features and best practices.
- Namespace structure makes the project very modular.
- Bugs in Clojure code can be **hard** to diagnose, beware of laziness!

Present

Examples

```
(use 'lobos.connectivity)
```

```
(def pg
  {:classname   "org.postgresql.Driver"
   :subprotocol "postgresql"
   :user        "test"
   :password    "test123"
   :subname     "//localhost:5432/test"})
```

```
(open-global pg)
(close-global)
```

||

```
(open-global :pg pg)
(close-global :pg)
```

You can specify a connection map (or global connection name) as the first argument of an *action* or use the global default connection.

Examples

All elements are constructed using functions or macros that yield Clojure data structures and thus are composable. Here we define some helpers:

```
(use 'lobos.schema)


(defn surrogate-key [table]
  (integer table :id :auto-inc :primary-key))


(defmacro tabl [name & elements]
  `(-> (table ~name (surrogate-key))
       ~@elements))


(defn refer-to [table ptable]
  (let [cname (-> (->> ptable name butlast (apply str))
                  (str "_id")
                  keyword)]
    (integer table cname [:refer ptable :id
                                     :on-delete :set-null])))
```

Examples

With the previously defined custom functions and macros we can define a more complex schema with less code...

```
(use 'lobos.core)


(defschema sample-schema :lobos

  (tabl :users
    (varchar :name 100 :unique)
    (check :name (> :length/name 1)))

  (tabl :posts
    (varchar :title 200 :unique)
    (text :content)
    (refer-to :users))

  (tabl :comments
    (text :content)
    (refer-to :users)
    (refer-to :posts)))
```

Examples

Schemas created with `defschema` are functions returning an abstract schema!

```
(create-schema sample-schema)

(set-default-schema sample-schema)

(drop (table :posts) :cascade)

(drop-schema sample-schema :cascade)
```

The cascade behavior works even in SQL Server which doesn't support that feature!

Dropping a schema created with `defschema` makes the defined function return `nil`, so just pretend we didn't evaled that line!

Warning: all uses of exclamation points in this slide are completely superfluous!

Examples

```
(alter :add (table :users (text :bio)))


(alter :add (table :users
              (text :location)
              (text :occupation)))


(alter :add (table :posts
              (check :content_limit
                     (< :length/content 1337))))


(alter :modify (table :users
                 (column :location
                         [:default "somewhere"])))


(alter :drop (table :users (column :occupation)))


(alter :rename (table :users
                 (column :location :to :origin)))
```

Any questions yet?

Do we need a break?

Project Structure

- Frontend
  - `lobos.core`
  - `lobos.schema`
  - `lobos.connectivity`
- Backend
  - `lobos.compiler`
  - `lobos.ast`
  - `lobos.backends.*`
- Analyzer
  - `lobos.metadata`
  - `lobos.analyzer`

Project Structure

$$\begin{tikzcd}
\text{connectivity} & \text{core} \arrow[l] \arrow[dl] \arrow[d] \arrow[ddr] & * \arrow[d] \\
\text{analyzer} \arrow[r] \arrow[d] & \text{schema} \arrow[d] & \text{utils} \\
\text{metadata} & \text{ast} & \text{compiler} \arrow[l] \\
 & \text{backends} \arrow[u] \arrow[ur] \arrow[ul, dashed] \arrow[uul, dashed] &
\end{tikzcd}$$

Architecture

- Actions yield Abstract Schema Definitions
- Schema protocols build a AST(s) from ASDs
- The compiler (`compile` multi-method) generates the appropriate SQL statements from an AST.
- The analyzer constructs an abstract schema ASD from a real schema.

Architecture

```
(defrecord Table [name columns constraints options]
  Alterable Creatable Dropable

  (build-alter-statement [this action db-spec]
    (let [elements (map #(build-definition (second %) db-spec)
                        (concat columns constraints))]
      (for [element elements]
        (AlterTableStatement.
         db-spec
         name
         action
         element))))

  (build-create-statement [this db-spec]
    (CreateTableStatement.
     db-spec
     name
     (map #(build-definition (second %) db-spec)
          (concat columns constraints))))

  (build-drop-statement [this behavior db-spec]
    (DropStatement. db-spec :table name behavior)))
```

Architecture

```
(def backends-hierarchy
  (atom (-> (make-hierarchy)
            (derive :h2 ::standard)
            (derive :mysql ::standard)
            (derive :postgresql ::standard)
            (derive :sqlite ::standard)
            (derive :sqlserver ::standard))))

(defmulti compile
  (fn [o]
    [(-> o :db-spec :subprotocol (or ::standard) keyword)
     (type o)])
  :hierarchy backends-hierarchy)

...

(defmethod compile [::standard AutoIncClause] [_]
  "GENERATED ALWAYS AS IDENTITY")

...

(defmethod compile [:mysql AutoIncClause] [_]
  "AUTO_INCREMENT")
```

Future

Migrations

Objectives:

- Could be written manually, like in `ActiveRecord::Migration`
- Could be automatically generated when actions are entered at the REPL
- Also automatically generated when declaratively changing a schema definition (see next slide)

Declarative Schema Manipulation

Say we are working on a database built from the following schema:

```
(defschema sample-schema :lobos

  (table :users
    (varchar :name 100 :unique)
    (check :name (> :length/name 1)))
...
```

Instead of imperatively issuing alter statements, wouldn't it be great to just change it?

```
(defschema sample-schema :lobos

  (table :users
    (varchar :name 100 :unique)
    (text :bio)
    (check :name (> :length/name 1)))
...
```

I need your help!

Visit the project's github page:

https://github.com/budu/lobos

Any questions suggestions, comments or insults?

Links

Website:

http://budu.github.com/lobos/

Code Repository:

https://github.com/budu/lobos

Issue Tracker:

https://github.com/budu/lobos/issues

Wiki:

https://github.com/budu/lobos/wiki

Acknowledgments

Thanks (in alphabetical order) to:

- James Reeves (for Hiccup)
- Lau B. Jensen (for the old and new ClojureQLs)
- Meikel Brandmeyer (for working with me on the old ClojureQL)
- Michael Fogus (for Marginalia and "The Joy of Clojure" book)
- Phil Hagelberg (for Leiningen)
- Rich Hickey (for Clojure)
- Stephen C. Gilardi (for clojure.contrib.sql)

Also thanks to all Clojure contributors and to its fantastic community!

The End!