REPORT:

STORING RELATIONAL DATABASE IN A KEY/VALUE STORE

(like Redis/MemcacheDB)

version 0.2 (approved)

Also available on https://github.com/pleomax00/relationalredis

Author: **Shamail Tayyab**
2010-544-047
M.Tech (Computer Science)
V Semester, Jamia Hamdard
Dated: 7th August 2012

CHAPTER 1: INTRODUCTION

Relational database are very popular since 1970s and are extremely useful for writing a data storage solution when the queries are unknown and the questions are asked at the run time, but relational databases scale very poorly when the data grows to a large scale.

To overcome this problem, we tend to use key/value stores and start denormalizing data. This study reveals on how can one store a data which can be represented in a key/value store, but still looks very relational and support the operations that relational algebra can do.

CHAPTER 2: RELATIONAL DATABASE

2. Relational Database

A relational database is a collection of data items organized as a set of formally described tables from which data can be accessed easily. A relational database is created using the relational model. The software used in a relational database is called a relational database management system (RDBMS). A relational database is the predominant choice in storing data, over other models like the hierarchical database model or the network model.

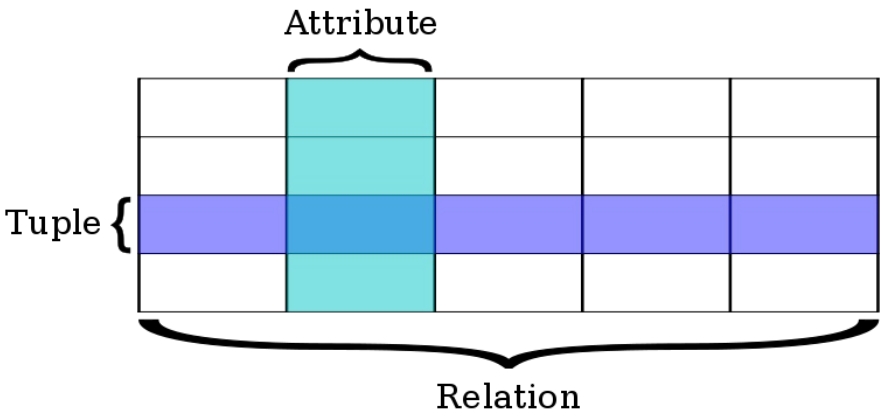

Fig 1. Visual Representation of a Relation

2.1 Relations or Tables

A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints. The relational model specifies that the tuples of a relation have no specific order and that the tuples, in turn, impose no order on the attributes. Applications access data by specifying queries, which use operations such as select to identify tuples, project to identify attributes, and join to combine relations. Relations can be modified using the insert, delete, and update operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting. It is necessary for each tuple of a relation to be uniquely identifiable by some combination (one or more) of its attribute values. This combination is referred to as the primary key.

2.2 Base and derived relations

In a relational database, all data are stored and accessed via relations. Relations that store data are called "base relations", and in implementations are called "tables". Other relations do not store data, but are computed by applying relational operations to other relations. These relations are sometimes called "derived relations". In implementations these are called "views" or "queries". Derived relations are convenient in that they act as a single relation, even though they may grab information from several relations. Also, derived relations can be used as an abstraction layer.

2.2.1 Domain

A domain describes the set of possible values for a given attribute, and can be considered a constraint on the value of the attribute. Mathematically, attaching a domain to an attribute means that any value for the attribute must be an element of the specified set. The character data value 'ABC', for instance, is not in the integer domain. The integer value 123, satisfies the domain constraint.

2.3 Constraints

Constraints make it possible to further restrict the domain of an attribute. For instance, a constraint can restrict a given integer attribute to values between 1 and 10. Constraints provide one method of implementing business rules in the database. SQL implements constraint functionality in the form of check constraints. Constraints restrict the data that

can be stored in relations. These are usually defined using expressions that result in a boolean value, indicating whether or not the data satisfies the constraint. Constraints can apply to single attributes, to a tuple (restricting combinations of attributes) or to an entire relation. Since every attribute has an associated domain, there are constraints (domain constraints). The two principal rules for the relational model are known as entity integrity and referential integrity. (“Referential integrity is the state in which all values of all foreign keys are valid. Referential integrity is based on entity integrity. Entity integrity requires that each entity have a unique key. For example, if every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the parent key of the table. To ensure that the parent key does not contain duplicate values, a unique index must be defined on the column or columns that constitute the parent key. Defining the parent key is called entity integrity”)

2.4 Primary keys

A primary key uniquely defines a relationship within a database. In order for an attribute to be a good primary key it must not repeat. While natural attributes are sometimes good primary keys, surrogate keys are often used instead. A surrogate key is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information about students at a school they might all be assigned a student ID in order to differentiate them). The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a tuple. Another common occurrence, especially in regards to N:M cardinality is the composite key. A composite key is a key made up of two or more attributes within a table that (together) uniquely identify a record. (For example, in a database relating students, teachers, and classes. Classes could be uniquely identified by a composite key of their room number and time slot, since no other class could have exactly the same combination of attributes. In fact, use of a composite key such as this can be a form of data verification, albeit a weak one.)

2.5 Foreign key

A foreign key is a field in a relational table that matches the primary key column of another table. The foreign key can be used to cross-reference tables. Foreign keys need not have unique values in the referencing relation. Foreign keys effectively use the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. A foreign key could be described formally as: "For all tuples in the referencing relation projected over the referencing attributes, there must exist a tuple in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."

CHAPTER 3: KEY VALUE STORES

3. Key/Value (or NoSQL) Stores

In computing, NoSQL (mostly interpreted as "not only SQL") is a broad class of database management systems identified by its non-adherence to the widely used relational database management system model, that is NoSQL databases are not primarily built on tables, and as a result, generally do not use SQL for data manipulation.

The following characteristics are often associated with a NoSQL database:

It does not use SQL as its query language

NoSQL database systems rose alongside major internet companies, such as Google, Amazon, and Facebook, which had significantly different challenges in dealing with huge quantities of data that the traditional RDBMS solutions could not cope with (although most of Facebook's infrastructure is based on MySQL databases and so is Twitter's). NoSQL database systems are developed to manage large volumes of data that do not necessarily follow a fixed schema. Data is partitioned among different machines (for performance reasons and size limitations) so JOIN operations are not usable.

It may not give full ACID guarantees

Usually only eventual consistency is guaranteed or transactions limited to single data items. This means that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system.

It has a distributed, fault-tolerant architecture

Several NoSQL systems employ a distributed architecture, with the data held in a redundant manner on several servers. In this way, the system can easily scale out by adding more servers, and failure of a server can be tolerated. This type of database typically scales horizontally and is used for managing big amounts of data, when the performance and real-time nature is more important than consistency (as indexing a large number of documents, serving pages on high-traffic websites, and delivering streaming media).

NoSQL database systems are often highly optimized for retrieve and append operations and often offer little functionality beyond record storage (e.g. key-value stores). The reduced run time flexibility compared to full SQL systems is compensated by significant gains in scalability and performance for certain data models.

CHAPTER 4: SCOPE OF WORK

4. Scope of Work

To research on the following topics and formulate concepts on:

- Getting to store a tabular data in a key/value store.
- Storing Foreign Key (one to many) relationships in the store.
- Storing many to many relationships in the store.
- Searching on a field.
- Indexing a field.
- Composite key indexes.

If time permits, create an ORM (Object Relational Mapper) for Redis (a key/value store) for python.

CHAPTER 4: SOURCE CODE

```
#! /usr/bin/env python

# @author: Shamail Tayyab
# @desc: Relational Model Store in Redis.
# @timestamp Wed Oct  8 20:21:37 IST 2012
# @filename: redismodels.py

import inspect
import redis

r = redis.Redis ()
r.flushall ()

class classproperty(object):
    def __init__(self, getter):
        self._getter = getter

    def __get__(self, instance, owner):
        return self._getter(owner)

class RField ():
    required = False
    default = None

    def __init__ (self, *k, **kw):
        if kw.has_key ("required"):
            self.required = kw['required']
        if kw.has_key ("default"):
            self.default = kw['default']

class StringField (RField):
    pass

class IntField (RField):
    pass

class ForeignKey (RField):
    def __init__ (self, *k, **kw):
        RField.__init__ (self, *k, **kw)
        self.relation = k[0]

class RModel (object):

    id = IntField ()

    keyvals = {}
    locals = []

    def __init__ (self, *k, **kw):
        self.newobj = True
        self.keyvals = {}
        self.locals = []
        self.reinit ()
        for i in self.locals:
```

```
            fieldobj = object.__getattribute__(self, i)
            if kw.has_key (i):
                self.keyvals[i] = kw[i]
            else:
                if fieldobj.required == True:
                    if fieldobj.default is not None:
                        self.keyvals[i] = fieldobj.default
                    else:
                        raise Exception ("Need a default value for %s" % (i))

    def from_id (self, id):
        self.seq = int(id)
        self.newobj = False
        return self

    def reinit (self):
        inspect.getmembers (self)

    def validate (self):
        if kw.has_key (name):
            self.keyvals[name] = kw[name]
        elif obj.default is not None:
            self.keyvals[name] = obj.default
        else:
            if obj.required:
                raise AttributeError ("This field is required")

    @property
    def classkey (self):
        return 'rmodel:%s' % (self.__class__.__name__.lower ())

    def sequence (self):
        seq_av_at = "%s:__seq__" % (self.classkey)
        seq = r.incr (seq_av_at)
        return seq

    def prepare_key (self, key, for_seq):
        r_key = "%s:%d:%s" % (self.classkey, for_seq, key)
        return r_key

    def save (self):
        if self.newobj:
            using_sequence = self.sequence ()
            self.keyvals['id'] = using_sequence
            self.seq = using_sequence
        else:
            using_sequence = self.seq
        for key, val in self.keyvals.items ():
            r_key = self.prepare_key (key, using_sequence)
            r.set (r_key, val)
        self.keyvals = {}
        self.newobj = False

    @classproperty
```

```
    def objects (self):
        return InternalObjectList (self)

    def __getattribute__ (self, attr):
        attrib = object.__getattribute__(self, attr)
        if not isinstance (attrib, RField):
            return attrib
        if attr not in self.locals:
            self.locals.append (attr)
        if self.newobj:
            if self.keyvals.has_key (attr):
                return self.keyvals[attr]
            else:
                fieldobj = object.__getattribute__(self, attr)
                return fieldobj.default

        answer = r.get (self.prepare_key (attr, self.seq))
        fieldobj = object.__getattribute__(self, attr)
        if answer == None:
            answer = fieldobj.default
        else:
            if isinstance (fieldobj, ForeignKey):
                fkey = r.get (self.prepare_key ('__relationfor__', self.seq))
                cls = globals ()[fkey]
                return cls.objects.get (id = answer)

        return answer

    def __setattr__ (self, attr, val):
        try:
            attrib = object.__getattribute__(self, attr)
        except AttributeError:
            object.__setattr__ (self, attr, val)
            return

        if not isinstance (attrib, RField):
            object.__setattr__ (self, attr, val)
            return

        if isinstance (attrib, ForeignKey):
            self.keyvals[attr] = val.id
            self.keyvals['__relationfor__'] = attrib.relation
        else:
            self.keyvals[attr] = val


class InternalObjectList (object):

    def __init__ (self, classfor):
        self.classfor = classfor

    def get_by_id (self, id):
        clsfor_obj = self.classfor()
        clsfor_obj.from_id (id)
```

```
        return clsfor_obj
        return
        for name, obj in inspect.getmembers (clsfor_obj):
            if isinstance (obj, RField):
                key = clsfor_obj.prepare_key (name, int(id))

    def get (self, *k, **kw):
        if kw.has_key ('id'):
            return self.get_by_id (kw['id'])

class Profile (RModel):
    fbid = StringField ()

class User (RModel):
    username = StringField (required = True)
    first_name = StringField (required = True)
    last_name = StringField ()
    password = StringField (required = True)
    email = StringField (required = True)

"""
# Testcase:
class FK (RModel):
    name = StringField ()

class Test (RModel):
    username = StringField ()
    password = StringField ()
    rel = ForeignKey ('FK')
    defa = StringField (default = 'a')
    req = StringField (required = True, default = 'abc')


fk = FK (name = 'abc')
fk.save ()

print "FKID:", fk.id

t = Test (username = "u", password = "p")
t.rel = fk
t.save ()
print t.id
k= t.rel
print "Naaam:", k.name


#t.username = "new"
#t.save ()

#t = Test ()
#t.username = 22
for i in r.keys ():
   print i, r.get (i)
"""
```