Key/Value Relational Mapping: A Concise Introduction

By Logan Gore

Abstract

The third most time-consuming task faced by most programmers is object/relational mapping (O/RM). The need for O/RM arises from the object/relational impedance mismatch that occurs when attempting to connect a relational table to a programming language's in memory objects [Fowler]. A SQL key/value architecture can help boost developer productivity and break this dependency on a custom data schema by offering an elastic approach to storing and retrieving objects from a relational table.

The two popular database models

The Relational Model

The relational model was first published in 1970 by Edgar Codd of IBM. In terms of data representation, a relational database must consist of two-dimensional tables containing rows and columns. When a row and a column intersect, that cell must contain an atomic value (a value such that sub-values cannot result from its division). Tables will contain primary keys which for each row will act as a unique identifier [Taylor].

Relational databases are flexible, reliable, and easy to understand due to an abundance of resources. The relational model, by nature, makes possible the powerful reporting that fuels many of today's data-driven solutions. The modern relational database management system (RDBMS) is optimized to provide complex data reporting under heavy loads. Most of all, the relational model is successful because familiar SQL provides an efficient and expressive means of communicating with databases.

For most developers the biggest cost to absorb from the relational model is object/relational mapping (OR/M). In general, the time spent on OR/M increases with the number of tables and objects that require mapping.

The NoSQL Model: Schema-less storage

Where a classic relational database has a set of tuples sharing the same attributes –the schema– NoSQL refuses such relations. "A NoSQL database is characterized by the lack of schema, the lack of a structured query language, and an often distributed and redundant architecture. NoSQL databases belong to three main families: document stores, key/value store, and object databases."[Esposito]

The NoSQL model has become especially popular for use with large distributed systems on the web. Easy clustering/sharding make the NoSQL model highly scalable. Large documents and objects can be accessed easily and stored in one area rather than spread over multiple tables.

When it comes to reporting capabilities the NoSQL model cannot compete with a relational database [Taylor]. The lack of a structured query language makes it difficult to switch NoSQL products without retraining and refactoring. In many cases data mapping will still need to be performed as well as heavy parsing if document stores are involved. The very nature of the NoSQL model makes it difficult to develop and evolve into the standard that the relational model has become.

The Key/Value Model in the RDBMS

NoSQL usually comes to mind when people think about key/value databases. But the relational database management system (RDBMS) is a good candidate for key/value storage, too (and even document storage). The popularity of relational databases and SQL makes the relational model a faster vehicle to the world of key/value storage. Rather than learning a new product and a new environment, developers can choose their familiar SQL platform as their new key/value store and have the option of integrating it into their existing relational database.

Key/Value Relational Mapping

Schema

In a relational model, the data schema (the table) is built to store a particular object where column names usually coincide with object properties. Figure 1 shows a table to store a simple *User* object:

Figure 1

ID	**FirstName**	**LastName**	**Email**	**ZIP**
1	Logan	Gore	logan@...	98326

The key/value model takes a *schema-blind* approach. An object's properties are stored in a database table as a collection of rows. A *Key* column holds the property name while its corresponding value is stored in a *Value* column. To provide clean associations between objects and their atomic properties, an

ID column can be used. Object data stored in this type of schema can therefore be queried by *ID*, returning all of the rows related to it (its properties), or in a more granular manner using *Key* and *Value* data as criteria. Figure 2 shows the *User* object from Figure 1 stored in a key/value table.

Figure 2

ID	Key	Value
1	FirstName	Logan
1	LastName	Gore
1	Email	logan@...
1	ZIP	98326

The primary advantage of the *schema-blind* table model is its elasticity. If a new property is added to an object during development, a new row is simply added to the table to hold the new property data. Developers no longer have to manage both their object schema and their data schema. The key/value architecture, by its nature, also allows for different kinds of objects to be stored in the same table.

Relationships

How might relationships be represented in a key/value database table? First, realize the limitations of the architecture, and work from there. Simple relationships (1…1, 1…n, FK) can easily be represented using an additional column in the key/value table. Figure 3 shows a key/value table with a *Parent* column to represent simple hierarchies. Child objects can then be queried by their *Parent* value.

Figure 3

ID	Key	Value	Parent
654981	Name	John Smith	89844
654981	Email	johnSmith@...	89844

Distinguishing between object types in the table

Where developers store different types of objects in the same table, being able to differentiate between them can help make the key/value model more expressive and avoid collisions between objects that could share the same *ID*. Figure 4 shows that adding an *Object* column is an easy route.

Figure 4

Object	ID	Key	Value
User	654981	Name	John Smith
User	654981	Email	johnSmith@...
Product	5151	Name	Widget
Product	5151	Price	1.99
Order	2222	Date	01/01/2015
Order	2222	Product	5151
Order	2222	User	654981

Data Types

Developers and DBAs usually look at the key/value model and ask (or growl) questions like "how are types handled?" They must first understand that the idea is promote a table that is generic –blind to schema and typing; a junk drawer of sorts.

In our experience, *ID* and *Key* columns work well as *varchar*. *Text* type is usually the best candidate for the *Value* column due to all the possible types of data being stored and their unknown lengths or sizes.

Casting should be handled in the programming language. However, in some cases developers may need to have the database perform casting operations dynamically through SQL.

Mapping to the key/value schema

The most obvious tool developers have to store and access key/value data in a programming language is a dictionary or hash table. These types of collection objects map well to a key/value database table. Nevertheless, more extravagant solutions can be created in languages like C# and Java that provide clean implementation of generics and reflection where instances of custom classes can be saved and loaded in the mapper.

Hardcoded SQL for purely relational systems is often frowned upon because of its maintainability issues. This is mostly a product of maintaining independent object and data schemas (the dual-schema problem). The fixed data schema of a key/value table makes mapping an easier task. A generic SQL *insert* statement can therefore be implemented for simple row level inserts and looped over for each object property.

Figure 5 shows the starting of a simple parameterized insert statement in C#. Notice that the column names are prefixed with *R_* , this is simply to avoid collisions with reserved words in many of the DB vendors.

Figure 5

```
public void Insert(string tableName, string objectName, string ID, string key, string value,
string parent)
{
	string sql ="INSERT INTO " + table + "(R_Object,R_ID,R_Key,R_Value,R_Parent)
	VALUES(@object,@id,@key,@value,@parent);";

	//...handle connection, command and parameters
}
```

What is a key/value relational mapper (K/VRM)?

A key/value relational mapper is a tool that is concerned with mapping data to key/value formatted rows in a relational database table.

To a developer, a K/VRM is a tool that allows for simple persistence and querying of objects in a database. Since the K/VRM can run on the familiar relational (SQL) database, developers may not need to learn a new language or DBMS, they only need to understand the behavioral differences in working with a key/value model. K/VRMs can be designed to accept responsibility for writing SQL, handling database commands and connections, and mapping objects to their generic structure.

How is a K/VRM different from an object/relational mapper (O/RM)?

Fundamentally, K/VRM is distinguishable from object/relational mapping (O/RM) in the sense that K/VRM is concerned only with mapping key/value pairs to key/value rows, rather than objects to custom rows. Key/value relational mappers, essentially, are only storing key/value pairs (properties) as rows that are linked to an object by an ID field. The object schema is of no importance to the K/VRM.

K/VRMs can show advantages over object/relational mappers (O/RMs) because K/VRMs don't require the developer to conceptualize, create, or maintain a database schema for each object. Objects that have an elastic or dynamic schema, whether due to refactoring or design, work well in a "schema-blind" key/value structure. These advantages keep developers productive instead of fighting with the most error prone and time consuming aspects of developing data-driven applications.

Limitations

The key/value model is not capable of solving the architectural, behavioral, or structural problems that stem from an object/relational impedance mismatch. However, the key/value model can effectively manage the *dual schema* problem as it is encountered in most applications.

After familiarizing yourself with how a key/value store can be applied to a SQL database, it will become apparent that the model has serious reporting limitations. For this reason, relational key/value stores

should be used for simple tasks such as saves, loads, and basic queries. Most apps, however, only need basic persistence and will benefit from the productivity of a key/value model.

Conclusion

Favoring generic design is axiomatic in an object-oriented world. It improves software quality and productivity full-circle. Most of all it saves money. The K/VRM is a great way to increase developer productivity and reduce errors in simple CRUD apps.

About the Author

Logan Gore is a software engineer who has published several Open Source projects for C#/.NET and papers on current industry topics such as enterprise architecture, cryptography, and Big Data. Logan is currently involved in cloud, game, and mobile development.

loganlgore@gmail.com

References

[Esposito]
Esposito, Dino. *Programming Microsoft ASP.NET 4. 2011*. Microsoft Press.

[Fowler]
Fowler, Martin. *Patterns of Enterprise Application Architecure*. 2003. Addison-Wesley.

[Taylor]
Taylor, Allen. *SQL All-in-One for Dummies, 2nd Ed*. 2011. John Wiley & Sons, Inc.