Hannes Voigt, Patrick Damme, Wolfgang Lehner

Enjoy FRDM - play with a schema-flexible RDBMS

Erstveröffentlichung in / First published in:

2015 IEEE 31st International Conference on Data Engineering. Seoul, 13.-17.04.2015. IEEE, S. 1460-1463. ISBN 978-1-4799-7964-6.

DOI: http://dx.doi.org/10.1109/ICDE.2015.7113401

Final edited form was published in "2015 IEEE 31st International Conference on Data Engineering. Seoul 2015", S. 1460-1463, ISBN 978-1-4799-7964-6
http://dx.doi.org/10.1109/ICDE.2015.7113401

Enjoy FRDM – Play with a Schema-flexible RDBMS

Hannes Voigt, Patrick Damme, Wolfgang Lehner
Database Technology Group, Technische Universität Dresden
01062 Dresden
{firstname.lastname}@tu-dresden.de

***Abstract*—Relational database management systems build on the closed world assumption requiring upfront modeling of a usually stable schema. However, a growing number of today's database applications are characterized by self-descriptive data. The schema of self-descriptive data is very dynamic and prone to frequent changes; a situation which is always troublesome to handle in relational systems. This demo presents the relational database management system FRDM. With flexible relational tables FRDM greatly simplifies the management of self-descriptive data in a relational database system. Self-descriptive data can reside directly next to traditionally modeled data and both can be queried together using SQL. This demo presents the various features of FRDM and provides first-hand experience of the newly gained freedom in relational database systems.**

I. Introduction

Traditional relational database management systems require modeling the mini world that should be represented before the database can be created and populated. Database development methods guide developers through the laborious process of requirement analysis, conceptional modeling, and physical modeling that yields the overall database schema. The underlying assumption is that the mini world to be represented is fairly stable and can be adequately modeled in a rigid database schema. Unfortunately, a growing number of application areas do not fulfill this assumption anymore. Application areas such as investigative analytics, data spaces, pay-as-you-go data integration are very dynamic and prone to schema changes. Data such as marketing campaign data, social data, internet log data, product catalogs data, and medical examination data are characterized by a very flexible, descriptive schema rather than an upfront modeled, prescriptive schema [1]. Such data is hard to model upfront and causes developers a headache when it has to be stored in relational databases. The recent boom of the flexible, descriptive data model JSON in these areas gives evidence of an increasing need for schema flexibility.

Relational systems are still accounting for 90 % of the information systems in larger enterprises [2]. Their prescriptive nature makes the managing of data with a flexible schema a troublesome business. To cope with flexible schemas, developers use workarounds such as universal tables or vertical schemas, which require complex, non-standardized, and often inefficient mapping layers. Another common solution is to move completely to more flexible data models such as RDF, XML, or JSON, which leads to polyglot persistence [3] and complicates data management in general and specifically the interoperability with existing relational systems. The large number of projects still taking these extra efforts shows that there is a real demand for first-class support of self-descriptive data and flexible schemas in relational database management systems.

In this demo we present the relational database system FRDM (speak [ˈfriːdəm]). FRDM is based on the open source database system H2 [4]. FRDM offers schema-flexible relational tables allowing the direct management of self-descriptive data in a relational database system. In FRDM, self-descriptive data can reside directly next to traditionally modeled data and both can be queried together using SQL. Flexible constraints allow limiting the schema flexibility on a fine-grained level, so that traditionally modeled data and self-descriptive data can coexist even within a single entity. With a comprehensive set of prepared examples, we will walk the visitor through the different features of FRDM. The visitor can also play freely with the system to get a first-hand experience and directly enjoy the newly gained freedom.

The aim of the demo is to convey the idea that flexible schemas and the relational data model are not contradicting each other. FRDM demonstrates that the relational data model can very well be extended and adapted to descriptive data management including flexible schemas and irregular structured data. We want to provide the experience of descriptive data management in relational fashion. Furthermore, we hope to spark more research in that direction.

II. FRDM

FRDM is a relational database system with standard relational data representation and processing capabilities. However, three substantial features of FRDM mark the distinction and turn FRDM into a flexible relational database management system for self-descriptive data: (1) flexible tables, (2) flexible entity domains, and (3) flexible constraints. We introduce all three briefly in the following subsections.

A. Flexible Tables

FRDM's first striking feature is that it centers data representation around tuples rather than around relations or tables. The flexible tables offered by FRDM are a mere container for tuples. Flexible tables do not prescribe any schema. For example assume we want to create a product catalog table where we cannot nail down the dynamic variety of product features in advance. In FRDM, we can create such a product table simply without defining any columns.

TABLE I. PRODUCT TABLE AFTER SOME INSERTS.

NAME	RESOLUTION	APERTURE	SCREEN	CAPACITY
Sony DSC-RX10	*20.0*	*2.8*	n/e	n/e
Samsung Galaxy S4	*13*	n/e	*4.3*	n/e
LG 60LA7408	*Full HD*	n/e	*60*	n/e
Sandisk Cruzer	n/e	n/e	n/e	*32GB*

```
CREATE FLEXIBLE TABLE Products;
```

Columns come to a flexible table as data is inserted into the table. We can insert data with a regular SQL INSERT statement, additionally we have to declare the columns the new tuple instantiates. For instance, the statement

```
INSERT INTO Products
VALUE DOMAINS (name, resolution, aperture)
VALUES ('Sony DSC-RX10', 20.0, 2.8);
```

adds a tuple $(\textit{Sony DSC-RX10}, \textit{20.0}, \textit{2.8})$ with the schema $(\text{NAME}, \text{RESOLUTION}, \text{APERTURE})$ to the table. FRDM determines the technical type of the value from the literal in the DML statement. Alternatively, the user can explicitly declare the type after the column name in VALUE DOMAINS clause – analogously to a column definition in the traditional CREATE TABLE statement. In any case, FRDM allows polyglot technical typing, i.e., values with a different technical type in the same column.

The schema of a flexible table automatically evolves with INSERT and UPDATE statements and remains purely descriptive. We can add more tuples regardless of these tuples having a similar or completely different schema. Consider the following two statements.

```
INSERT INTO Products
VALUE DOMAINS (name, resolution, screen)
VALUES ('Samsung Galaxy S4', 13, 4.3),
       ('LG 60LA7408', 'Full HD', 60);
INSERT INTO Products
VALUE DOMAINS (name, capacity)
VALUES ('Sandisk Cruzer', '32GB');
```

Both statements are valid in FRDM and add data to the table. Along the way, FRDM automatically adds the columns SCREEN and CAPACITY to the table.

We can also add columns by updating individual tuples. To add a weight to the product *Sony DSC-RX10*, we issue the statement

```
UPDATE Products SET weight = 813
WHERE name = 'Sony DSC-RX10';
```

Again, FRDM automatically adds the column WEIGHT to the table. In similar fashion, we can remove individual values easily by updating them as not existing.

```
UPDATE Products SET weight = NOT_EXISTING
WHERE name = 'Sony DSC-RX10';
```

Note that FRDM does not automatically remove a column from a table since this would require checking whether any of tuples instantiates the column. Of course, the user can remove columns manually with an ALTER TABLE ... REMOVE VALUE DOMAIN statement.

Flexible tables can be queried as traditional tables with standard SQL. The query SELECT * FROM Products yields the result set shown in Figure I. Columns that are not part of a tuple's schema simply do not exist for that tuple (marked as n/e). It is important to understand that not existing columns carry a different semantic than NULL values. A NULL indicates an instantiated column where the value is currently unknown. FRDM strictly distinguishes between both cases.

On top of standard SQL, FRDM offers a set of additional expressions that help exploiting and dealing with the schema irregularity that flexible tables allow. As tuples may vary in the schema they instantiate, queries can filter for tuples instantiating certain columns. For instance, the query

```
SELECT * FROM Products WHERE screen IS EXISTING;
```

selects all tuples that instantiate the column SCREEN. IS EXISTING is a special predicate that evaluates to false if the column does not exist for a tuple. The fact of a not existing column propagates through all standard SQL expressions. Where a predicate results in non-existence, the evaluated tuple is not further processed. Where a projection expression results in non-existence, the resulting tuple will not instantiate the projected column. As an example consider the query

```
SELECT name || ' has a ' || screen || 'inch
screen' FROM Products WHERE aperture < 3;
```

which results in an empty result set. All tuples but the Sony camera have no APERTURE column and do not qualify for the predicate. In turn, the Sony camera has no SCREEN column, so that the projection results in a tuple not instantiating any column. Such empty tuples are excluded from any further processing.

FRDM features also expressions to handle different technical types in a column. Next to standard cast functions, FRDM offers the TYPENAME(<exp>) function to retrieve the type, the <exp> HASTYPE <typename> predicate to check for a type, and the CASETYPE clause to formulate type-sensitive expressions. Here is an example of a CASETYPE expression:

```
CASETYPE x WHEN INT THEN x+1
           WHEN VARCHAR THEN x || ' plus one'
           ELSE x END
```

B. Flexible Entity Domains

FRDM's second striking feature is the possibility to tag tuples with entity domains. This allows marking tuples with the concepts they belong to or the sources they come from. The tagging is done with DML statements. For instance, we can use the statement

```
UPDATE Products
CHANGE ENTITY DOMAINS ADD (Cameras, GPSs, Phones)
WHERE name = 'Samsung Galaxy S4';
```

Final edited form was published in "2015 IEEE 31st International Conference on Data Engineering. Seoul 2015", S. 1460-1463, ISBN 978-1-4799-7964-6
http://dx.doi.org/10.1109/ICDE.2015.7113401

TABLE II. PRODUCTS TAGGED WITH ENTITY DOMAINS.

	CAMERAS	GPSS	PHONES	TVS
Sony DSC-RX10	✓			
Samsung Galaxy S4	✓	✓	✓	
LG 60LA7408				✓
Sandisk Cruzer				
Canon EOS 6D	✓	✓		
Garmin Dakota 20		✓		
Nokia 108	✓		✓	

to tag the product *Samsung Galaxy S4* with entity domains CAMERAS, GPSS, and PHONES. We can also tag tuples during inserts. The statement

```
INSERT INTO Products
ENTITY DOMAINS (Cameras, GPSs)
VALUE DOMAINS (name, resolution)
VALUES ('Canon EOS 6D', 20);
```

inserts the product *Canon EOS 6D* and tags it with CAMERAS and GPSS. For the following, assume we have added more tuples and have tagged all tuples as shown in Table II.

FRDM allows to use tags in the `FROM` clause to directly address subsets of tuples in a flexible table. The statement

```
SELECT * FROM Cameras;
```

queries all cameras from the PRODUCTS table. Each entity domain belongs to one flexible table. FRDM automatically resolves the table for the entity domain CAMERAS. In case of name collisions, an entity domain has to be prefixed with the table name, e.g., `Products.Cameras`. To query combinations of entity domains, FRDM offers set operations directly within the `FROM` clause. For instance, the query

```
SELECT *
FROM Cameras INTERSECT GPSs MINUS Phones TRIM;
```

yields all non-phone cameras with GPS as shown in Table III. The `TRIM` clause at the end of the query removes all not instantiated columns in the result set. The complement of an entity domain in the whole flexible table can also be expressed easily, e.g., all non-camera products:

```
SELECT name FROM Products MINUS Cameras;
```

For the result of this query see Table IV.

Flexible tables and entity domains are completely interoperable with traditional tables. Assume we use a traditional table to store suppliers of the items in the `Products` table. We can then easily join this `Suppliers` table with the entity domain `Cameras` to get all camera suppliers:

```
SELECT DISTINCT s.name, s.address
FROM Suppliers s
     INNER JOIN SuppProd sp ON s.id = sp.suppId
     INNER JOIN Cameras c ON sp.productId = c.id;
```

TABLE III. NON-PHONE CAMERAS WITH GPS

NAME	RESOLUTION
Canon EOS 6D	*20*

TABLE IV. NON-CAMERAS PRODUCTS

NAME
LG 60LA7408
Sandisk Cruzer
Garmin Dakota 20

C. Flexible Constraints

As its third distinguishing feature, FRDM offers flexible constraints. Where traditional constraints epitomize the prescriptive nature of relational database management systems, flexible constraints adapt the descriptive nature of FRDM. Most importantly, flexible constraints can vary in their effect. While traditional constraints are always prohibitive, FRDM's flexible constraints can also be informative, warning, and hiding. For instance, the constraint

```
ALTER TABLE Products ADD CONSTRAINT
negativePrices
CHECK price >= 0 EFFECT WARNING;
```

issues a warning on every operation that manipulates prices to negative values; but the constraint does not forbid such an operation. The same constraint with the effect `HIDING` also permits all operations but additionally hides product tuples with a negative price from regular queries. Consequently, a query on `price < 0` returns an empty result set. However, hidden tuples can be queried or manipulated explicitly by using the constraint in the `WHERE` clause. We can use the query

```
SELECT * FROM Products
WHERE Products VIOLATES negativePrices;
```

to find all products with a negative price. Analogously, the DML statement

```
UPDATE Cameras SET price = price * -1
WHERE Products VIOLATES negativePrices;
```

makes all negative prices positive and these cameras become visible again to regular queries.

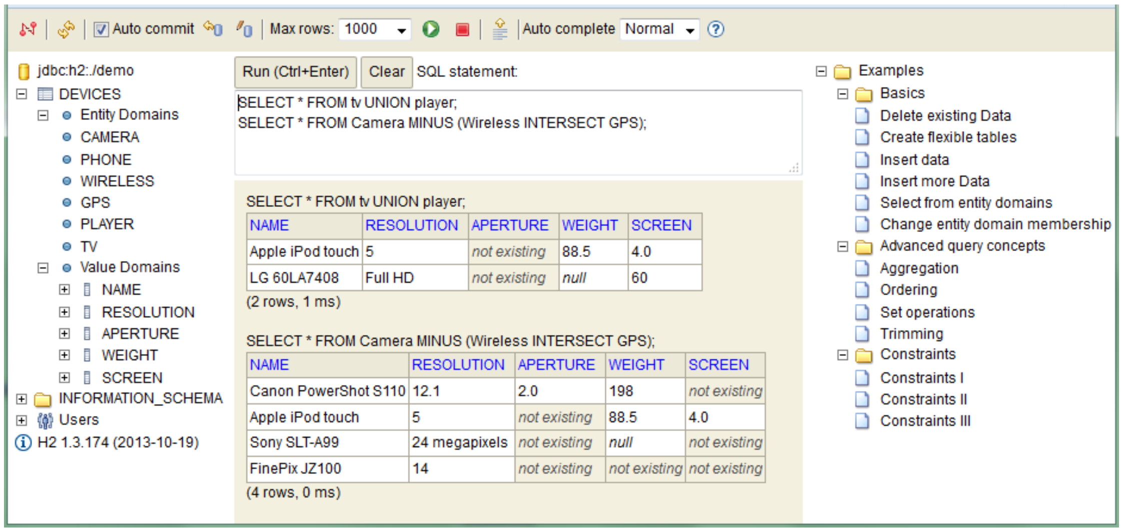
Entity domains and columns can be used to restrict to which tuples a constraint applies. As an example, the constraint

```
ALTER TABLE Products ADD CONSTRAINT
negativeScreen
ENTITY DOMAIN (Cameras UNION Phones)
CHECK screen > 0 EFFECT PROHIBITING;
```

prohibits a negative screen size for all cameras and phones that instantiate the `screen` column.

Next to standard check and key conditions, flexible constraints can also express structural conditions on the data. For instance, if we want to be warned about phones that are not cameras but have an aperture, we can create the constraint:

```
ALTER TABLE Products ADD CONSTRAINT noAperture
ENTITY DOMAIN (Phones MINUS Cameras)
CHECK Products NOT INSTANTIATES aperture
EFFECT WARNING;
```



```
SELECT * FROM tv UNION player;
SELECT * FROM Camera MINUS (Wireless INTERSECT GPS);
```

SELECT * FROM tv UNION player;

NAME	RESOLUTION	APERTURE	WEIGHT	SCREEN
Apple iPod touch	5	*not existing*	88.5	4.0
LG 60LA7408	Full HD	*not existing*	*null*	60

(2 rows, 1 ms)

SELECT * FROM Camera MINUS (Wireless INTERSECT GPS);

NAME	RESOLUTION	APERTURE	WEIGHT	SCREEN
Canon PowerShot S110	12.1	2.0	198	*not existing*
Apple iPod touch	5	*not existing*	88.5	4.0
Sony SLT-A99	24 megapixels	*not existing*	*null*	*not existing*
FinePix JZ100	14	*not existing*	*not existing*	*not existing*

(4 rows, 0 ms)

Fig. 1. Screenshot of the FRDM web interface.

The constraint "phones instantiating the column `ostype` also must instantiate the column `osversion`" is enforced by:

```
ALTER TABLE Products ADD CONSTRAINT osTypeVersion
ENTITY DOMAIN (Phones) VALUE DOMAIN (ostype)
CHECK Products INSTANTIATES osversion
EFFECT PROHIBITING;
```

Beyond the aspects discussed here, FRDM smoothly integrates flexible tables, flexible entity domains, and flexible constraints also with other fundamental database technologies such as transaction handling, indexing, and views.

III. FRDM Demo

The demo is based on the FRDM web interface, shown in Figure 1. It allows to connect to a FRDM database and submit DQL as well as DDL and DML statements. Statements are entered in the text area at the top; results are displayed on the canvas below the text area. Any changes to the schema of the database, e.g., new columns added to a flexible table by a DML operation, are reflected in the schema tree on the left. The right frame lists statements that we have prepared for the demo.

The prepared storyline of the demo takes the visitor through a database workflow of an online electronics dealer, who is maintaining a product catalog while the products rapidly evolve and the dealer is expanding its business. In the first half of the storyline, we set up a database consisting of regular tables for order and supplier information and flexible tables for the product information. Then, the product catalog is populated gradually with products. The product range evolves constantly and so does the schema of the product catalog. Concurrently, the database is queried for product information and order processing. Going through all these steps, the visitor will experience how FRDM simplifies the management of schema-flexible data by avoiding complex mapping layers or polyglot persistence.

While our dealer embraces the schema flexibility, it is also important to the dealer to keep the data in order while its business grows. In the second half of the storyline, we start to organize the products into entity domains. For each entity domain we set up flexible constraints that demand a minimum set of attributes to allow for a better online presentation of product features. This demonstrates that the approach of FRDM is not purely descriptive but has the flexibility to range anywhere between purely descriptive and purely prescriptive.

We plan to conduct the demo very interactively. The prepared storyline serves merely as the starting point for the demo. It allows us to quickly introduce the visitor to the various features of FRDM and demonstrate their benefits. We run the storyline adaptively as the discussion with the visitor evolves. At any time it is possible to deviate from the storyline, change statements or draw up completely new scenarios. The visitor can play freely with the system, as well. For very specific questions, we additionally have a very long list of detailed examples, which we used to test the system. Further, we are open to discuss with the visitor how we implemented the various features and are eager to hear the visitor's opinion in that regard. To foster such a discussion, we prepared also a handful of examples demonstrating the performance challenges that come with implementing such functionality.

With FRDM we want to show the feasibility of flexible schema data management in RDBMs and SQL while avoiding complicated mapping layers and polyglot persistence. More generally, we hope to demonstrate the benefits of integrating the advantages of descriptive and prescriptive data management in a single system.

References

[1] C. Monash, "Data model churn," DBMS2 Blog: http://www.dbms2.com/2013/08/04/data-model-churn/, Aug. 2013.

[2] M. L. Brodie and J. T. Liu, "OTM'10 Keynote: The Power and Limits of Relational Technology In the Age of Information Ecosystems," in *OTM'10*, vol. 6426, 2010.

[3] M. Fowler and P. Sadalage, "NoSQL Database and Polyglot Persistence," Personal Website: http://martinfowler.com/articles/nosql-intro-original.pdf, Feb. 2012.

[4] T. Mueller, "H2 Database," http://www.h2database.com, 2012.