WHAT IS A MULTIVALUE DATABASE

A Simpler Way to Store and Manage Complex Data

Abstract

Multivalue databases challenge relational dominance by storing complex data naturally. This approach simplifies application development and offers a powerful alternative for critical systems.

Martin Phillips

What is a MultiValue Database?

If you are an experienced database professional, you will be familiar with the relational database model. This model is widely taught in academic institutions, often with an implication that it is the only valid way in which a database can be constructed. The MultiValue database model successfully challenges this viewpoint and has significant advantages. With the growing NoSQL movement encouraging a move away from the relational database model, MultiValue is the obvious and well proven alternative.

The MultiValue model was created at about the same time as the relational model but has not received the recognition that it deserves, however, it is widely used in systems ranging from simple single user applications to the massive databases employed by some of the world's largest corporations. Perhaps one reason why it is not more widely known is that it rarely makes the news because MultiValue database projects tend be delivered on time and within budget. Developers who work with this model are usually so enthusiastic about it that it is surprising that it has remained unheard of by many database professionals.

One of the key features of the MultiValue model is that it enables information to be stored in the database in a form that more closely relates to the real world entities that the data represents than is possible in a relational database. It achieves this by relaxing one of the rules imposed by the relational model that otherwise complicates database design, increases application complexity, and hence raises development costs.

Before exploring the MultiValue data model, it is worth observing that this is a superset of the relational model. Everything that can be done in a relational database can be done in MultiValue but it is the ability to store multi-dimensional data that makes this model so powerful.

Database Tables

There are many ways to represent data in a database but the most common is as simple two dimensional tables, each containing data of a specific type. As an example, a very simple sales processing system might have just three tables storing the stock data (STOCK), customer details (CUSTOMERS) and orders placed (SALES).

The STOCK table that stores product stock levels might be structured as in the diagram below.

Part No	Description	Quantity	Price
003	Pen, blue	87	1.70
004	Pen, green	88	1.70
011	Pencil, black	171	0.30
013	Pencil, blue	13	0.30
051	Scissors, small	17	2.27
111	Desk lamp	3	17.50

Adopting some terminology, each row of this table is referred to as a **record** and the unique identifier for a row (in this case the part number) is referred to as the **key** or **record id**.

The Rules of Normalization

The Rules of Normalization were proposed by Ted Codd in 1970 and, although put forward as a discussion document, have been adopted by most relational database developers as fundamental design laws. The first three rules are the most significant and each step transforms the data to the next "normal form".

First Normal Form: Eliminate repeating data by creating a separate table for each set of repeating data.

By way of an example, consider our simple order processing system. Although we may find it easiest to think of an order as single entity formed from the date, customer number and a list of items and quantities, the first normal form requires us to store repeating data such as the list of items and quantities separately from the other data.

Second Normal Form: Separate tables for sets of values that apply to multiple records.

In practical terms, this means that we avoid storing the same information in multiple places, partly to save space but more significantly because changes to that data (e.g. a change of address) require only a single database update. It is important to distinguish truly duplicated data from distinct copies of the same data that could later change. For example, it is valid to copy the customers address from his account details into an order record because historic data for stored orders should not change if the customer later relocates.</p>

Third Normal Form: Eliminate fields that do not depend on the key.

A database record should only hold information that is dependent on the key. For example, the product descriptions should not be stored in the order record as these can be derived by using the product number to access the table of stock details. Conversely, the price should be copied from the STOCK table into the SALES table at the time of placing the order so that subsequent price changes do not affect the data recorded for past orders.

Discarding the First Rule

The second and third normal forms represent good design practice but the first normal form is largely an artificial and unnecessary step that was partly introduced to make the internal workings of the database software easier. The effect of applying this step is to move away from a database in which the data is stored in a form that closely models the real world entities that it is representing. In the MultiValue data model, the first rule of normalization is discarded (though there may be special occasions when it is useful to apply it).

We are probably all familiar with representing data in MultiValue form in our everyday lives. Consider the receipt from a shopping trip to the local supermarket. This will have non-repeating data such as the store name and the date of purchase but it will also have a list of the descriptions, quantities and prices of each item purchased. It would be unreasonable for the supermarket to produce a totally separate receipt for each item purchased and yet that is effectively what the first normal form forces us to do inside the database. Unless the supermarket is a MultiValue user, there is a good chance that their software breaks the data from the checkout into separate tables in the database and that the process of producing the receipt requires this data to be reassembled into a single entity. In a MultiValue database, the transaction is stored in the database as a single record that represents the entire purchase.

The MultiValue Data Model

Consider the SALES table. The rows of the table (records) each hold the data for an individual order. The columns, known as **fields** or **attributes**, hold information about the orders. Keeping this very simple, we might need columns to store the order date, a cross-reference to the CUSTOMERS table (probably the customer number) and a list of part numbers, quantities and prices.

Order No	Date	Customer	Product	Quantity	Price
12000	14 Jan 11	1632	101	3	3.50
12001	14 Jan 11	0783	106 210	1 2	1.75 4.00
12002	17 Jan 11	1373	201	8	8.30

The diagram above shows how the MultiValue model allows us to store the entire order as a single record instead of breaking the repeated data (product, quantity, price) into a separate table. As our orders become larger, the performance advantage of this model becomes more significant. Also, by reducing the number of tables, the application becomes simpler to understand and hence easier to develop and maintain.

Our SALES table has just six fields including the record id. Realistic databases are much more complex with, perhaps, over one hundred fields at times.

To fully understand the power of the MultiValue data model, it is useful to understand how this data is represented inside the database. All data is normally stored in character format rather than storing binary values for numeric items. The data in each field is concatenated to form a single character sequence that represents the entire record. To provide a way to know where the boundaries between the fields lie, we insert a special marker character called a **field mark** between each field. Similarly, where a field contains multiple values, these are separated by **value mark** characters. Thus order 12001 from the diagram above becomes

14 Jan 11 $F_{M\ 0783}$ $F_{M\ 106}$ $V_{M\ 210}$ $F_{M\ 1}$ $V_{M\ 4}$ $F_{M\ 1.75}$ $V_{M\ 4.00}$

Character data that is divided into fields and values is known as a **dynamic array**, a term that emphasizes that it is a list of items, each of which can be of variable format. Although dynamic arrays are most commonly used to represent database records, they are often used for other purposes within MultiValue applications.

There are several important things to note about this representation of our data record.

Firstly, note that the order number is not in the data. In any database system, we need a way to uniquely identify the records in the table, in this example by use of the order number. In the MultiValue data model, the unique identifier (known as the **record id** or **key**) is not considered to be part of the data but is a handle by which we access the data. Although it may at times be helpful to consider it as part of the record as in the diagram above, reading a record from the database returns only the data fields.

Secondly, notice that although there is a value by value relationship (an **association**) between the values in the product number, quantity and price fields, there is nothing in the stored data to show this. The structure and interpretation of the content of a record is defined by the table's **dictionary**.

Thirdly, by use of mark characters, it is possible to store truly variable length data. There will come a time when the customer numbers need to be expanded to five digits. With the MultiValue model, we can store these as variable length so that no changes are needed to the database when the number of digits increases. Use of mark characters also implies that the multivalued fields can hold any number of values. The data model imposes no restrictions.

The database may at times store data internally in a different form than would be used to present this data to a user. Dates are usually stored as a number of days from a reference point in time and fractional values are normally scaled into a lower unit to eliminate the decimal point. Our example record above becomes

15720 $F_{M\ 0783}$ $F_{M\ 106}$ $V_{M\ 210}$ $F_{M\ 1}$ $V_{M\ 2}$ $F_{M\ 175}$ $V_{M\ 400}$

The data model allows us to go one more level, breaking values into **subvalues**. Perhaps our order processing database needs to store the serial numbers of each part sold. We could add a further field that has a value for each part number, further divided into a subvalue for each serial number.

15720 $F_{M\ 0783}$ $F_{M\ 106}$ $V_{M\ 210}$ $F_{M\ 1}$ $V_{M\ 2}$ $F_{M\ 175}$ $V_{M\ 400}$ $F_{M\ 81}$ $V_{M\ 98}$ $S_{M\ 472}$

If this looks complex, take the dynamic array apart one layer at a time. First identify the fields, then identify the values in each field and then, if relevant, identify the subvalues in each value.

By storing our data in this form, our orders table is now a four dimensional structure that, once we understand the concept, is far simpler than the multi-table representation forced upon users of the relational model. Also, the additional processing needed to locate and extract items from the dynamic array is usually far less costly than merging data from separate tables as would be needed in a relational database. Note, however, that there is no reason why a MultiValue database cannot store fully normalised data if the developer so wishes.

So what are the mark characters? When the MultiValue data model was originally created, the upper half of the 8-bit character set was undefined. The designers of the data model chose to use the final five characters (251 to 255) as the marks.

	Char	Symbol	@VAR
Item mark	255	IM	@IM
Field mark	254	FM	@FM
Value mark	253	VM	@VM
Subvalue mark	252	SM	@SM
Text mark	251	TM	@TM

In the table above, the column headed Symbol is how the mark character is shown in dynamic arrays in the QM documentation. The column headed @var is a code that can be used in applications to reference the mark character.

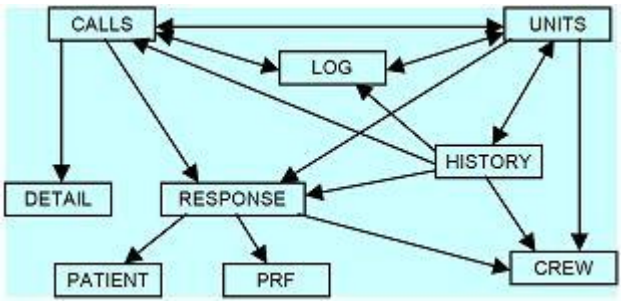
The **item mark** (sometimes known as the **record mark**) is used internally in some MultiValue systems but has little use by developers. The field (attribute), value and subvalue marks are as discussed above. The **text mark** is used in some places to mark positions where text data could be split over multiple lines.

The upper half of the 8-bit character set is now defined and tends to contain language specific characters. In Europe, for example, it contains the accented characters used by many languages. This poses a problem as, for example, the German u-umlaut (ü) has the character value that the MultiValue model assigns to the subvalue mark. It is, therefore, not possible to store, for example, the name Müller. This is clearly unacceptable in some situations and most MultiValue products, including QM, provide an option to work with an extended character set such as Unicode.

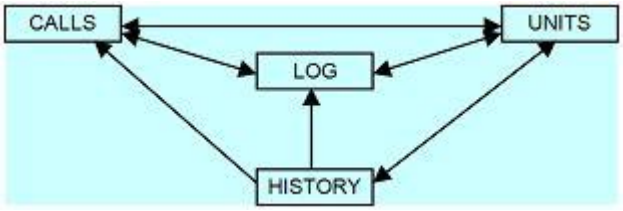
How Does MultiValue Simplify Applications?

As described above, by relaxing the first rule of normalization, the MultiValue model simplifies applications. But what does this mean in real terms?

The diagrams below represent part of a real application for controlling emergency ambulances (a critical role in which MultiValue is often used). If we model the call logging and dispatch elements of this application using the relational model, we end up with a minimum of nine database tables and fifteen inter-relationships.



Redesigning this with the MultiValue model yields an application with just four database tables and six inter-relationships.



As additional elements of the application are added, the complexity of the relational model rises rapidly whereas the MultiValue version tends to remain simple. Complexity gives rise to higher development and maintenance costs as well as greater risk of error.