Horizontal or Vertical? Beyond the Relational Database

Relational databases are the industry standard, even though specialist skills are required for designing them; whatever the expertise of the database designer, the result is always a compromise as the process of normalisation produces artificial constructs that cannot model the complexity of the multi-dimensional real world.

Typically, the data for any particular file or entity type are stored in a single fixed length contiguous record, with each type of data stored in a separate file. We might describe this type of record layout as “horizontal”.

Consider a simple Customer File example with the fields Customer Number, Customer Name, Address, Telephone Number, and Credit Limit.

Problems arise if any customer has more than one address or more than one telephone number or if, after the file has been created and put into production, an additional field, such as customer type, is required. Intellectually these are simple problems, but the changes are difficult to implement, and, having allowed for, say, two telephone numbers, you will inevitably come across a customer with three telephone numbers. Most customers may have one telephone number but, if you allow for three, then, in a large customer file, space is wasted in most records as the empty fields must be present but have a “null” value. With disk storage prices now so low, wasted space is no longer an issue, but the longer records do have an adverse impact on performance, which is an issue in many applications.

However, the most obvious problem is changing the layout of a file when repeating or additional fields are required.

An alternative solution is to put fields, or each attribute, in separate records, using a key field value, such as Customer Number, to link them together. These might then have the layout

Customer Number, Customer Name
Customer Number, Address
Customer Number, Telephone Number
Customer Number, Credit Limit

We might describe this file layout as “vertical”.

This document outlines the evolution of this concept which resulted in the creation of ERROS (**E**xpert **R**ealtime **R**elational **O**pen **S**ystem), a most powerful application creation system that allows incremental development of very complex applications without a detailed upfront specification, without any physical database design and mostly without programming. The ERROS Connectionist Database handles variable length records without the need for null values and is responsible for extremely rapid response times. ERROS is not a relational database and is unlike any other database management system. The concepts of ERROS have been successfully patented.

To ensure that the purpose of each attribute record listed above is clearly identified, an extra field, attribute number, could be put between Customer Number and the attribute data. If this were, say, five digits long, then 99999 different attributes could be defined for the file.

To allow repeating fields for, say, additional telephone numbers, a second extra field, attribute iteration number, should be put between the attribute number and the attribute data. This would be part of the key field. If this can store, for instance, a three digit number, then each attribute could have 999 iterations – any customer could have 999 telephone numbers if necessary, but no space is wasted where a customer only has one telephone number or even none.

This allows much greater flexibility than the fixed length “horizontal” record described earlier.

Sample records might be

KEY FIELD			USER DATA	
Record Number	Attribute Number	Record Id. Suffix	Record Data	Record Number
057692	00000	001	XYZ Company	57692
057692	00001	001	23 Acacia Avenue, Newtown	
057692	00002	001	0 1234 123456	
057692	00003	001	GBP 10000	

Fig. 1

If the records are indexed on the first three numeric fields, then adding new records, including newly defined attributes, will be very straightforward and the records will be retrieved by the database handler in the correct sequence.

Note that I have numbered the first attribute, the customer name, which is the identifying attribute, as attribute 00000. The reason for this is explained later.

If a new attribute is added, such as customer type, then this can be allocated a new attribute number and, as this is part of the key field, an extra record can be added without the need to change any of the existing records. The result would be

KEY FIELD			USER DATA	
Record Number	Attribute Number	Record Id. Suffix	Record Data	Record Number
057692	00000	001	XYZ Company	57692
057692	00001	001	23 Acacia Avenue, Newtown	
057692	00002	001	0 1234 123456	
056792	00003	001	GBP 10000	
056792	00004	001	retail	

Fig. 2

If the records for all attributes were the same length and put in the same file, the database handler would be simple to construct and response times would be much faster. If the data for one attribute cannot fit into one record, then more than one can be used.

The horizontal file layout does not allow history to be stored – if a customer's telephone number is changed, the old number is lost as it is overwritten with the new one.

If, with the vertical layout, a single byte dormant marker field is added to every record, then, to change a telephone number, the old telephone number record is marked as dormant, a new record is added for the new number, and the history is retained. Equally, multiple telephone numbers can be stored for any customers. The absence of a record indicates that there is no data, but, unlike null values, this takes no space. Records can be added at any time.

KEY FIELD			CONTROL DATA	USER DATA	
Record Number	Attribute Number	Record Id. Suffix	Dormant Marker	Record Data	Record Number
057692	00000	001		XYZ Company	57692
057692	00001	001		23 Acacia Avenue, Newtown	
057692	00002	001	D	0 1234 123456	
057692	00002	002		0 3123 456821	
056792	00003	001		GBP 10000	
056792	00004	001		retail	

Fig. 3

The first field in the key field in this example, is the customer number. We may know the customer's name, yet we probably won't remember the customer's number. We need a way of looking it up by typing in the customer name without searching the whole file. Name is a valid record identifier as are, for instance, date or date and time, or telephone number or postcode or address, order number, etc. ERROS also handles synonyms.

Name could be part of the key field to allow searching by name, but, if the name was, say, 64 characters long, then this index field would need to be 64 characters long and this is too large as part of the key field. However, by using a two stage, patented algorithm, this can be reduced to a perfectly acceptable 4 bytes. This would allow 999,999,999 customer records. The algorithm enables ERROS to map all records to locations on disk without the developer knowing where or in which file these are stored. It also allows records to be accessed in name sequence, and, for personal names, in the sequence of last name, first name(s).

We could use a similar structure for all our files. Once a database handler has been created that can handle the new structure, then the ability to allow multiple attribute iterations could be applied to all attributes in all files with no new database handler programming being required. The records could all be accessed by name.

Until the customer number has been established, it will be zero. If we add an additional compressed name identifier field as field 3, and include this as part of the index or key field, then the record layout might be -

KEY FIELD				CONTROL DATA	USER DATA	
Record Number	Attribute Number	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number
000000	00000	XYZ Company (compressed)	001		XYZ Company	57692
057692	00001	23 Acacia Avenue (Compressed)	001		23 Acacia Avenue, Newtown	
057692	00002	01234123456	001	D	0 1234 123456	
057692	00002	03123456821	001		0 3123 456821	
056792	00003		001		GBP 10000	
056792	00004		001		retail	

Fig. 4

Attributes can have one or more fields. They can also store unlimited text.

Note that, as the record identifier is now part of the key field, the second telephone number now has a unique identifier so its Record Id. Suffix is now 001. In addition, all telephone numbers for any customer are stored in number sequence within customer. If the names of the employees of the XYZ Company were included, they would be stored in the sequence of last name, first name(s) within the attribute "employees" and, as they would all be indexed, any employee name can be retrieved instantly.

Thus, if a user types in XYZ Company, as it is the identifying attribute, the algorithm will set the record and attribute numbers to zero and calculate the compressed identifier. As the records are indexed, the database handler will find the record immediately without having to search the database. If there are records with duplicate names, the database handler will retrieve them. If the user only types XYZ (or xyz), the database handler will retrieve all records whose names began with XYZ, such as XYZ Company, XYZ Group, XYZ Holdings, etc. Having accessed the record for the XYZ Company, their customer number is retrieved from the second data field. All attribute records for the XYZ Company can now be accessed as these are stored with the customer number as the first part of the key field and the database handler will read them all and pass them to the presentation layer for display. This could present them as one contiguous record so that users would not necessarily be aware that the data is stored in a different way from a horizontally organised database.

The database handler must also include a mechanism that allocates the next available number for new customers if none has been entered by the user.

As new customer identifier records are added, they will be stored in name sequence so that a list of customers is easily displayed. All attribute values for each customer are stored together so that performance is excellent. ERROS is highly scalable.

Users are not aware of the structure of the key field – they simply find the method of operation entirely logical and easy to use. The figure below shows how the records are stored (the gaps would not exist in the database).

KEY FIELD				CONTROL DATA	USER DATA	
Record Number	Attribute Number	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number
000000	00000	Smith, Fred (compressed)	001		Fred Smith	65737
000000	00000	Smith, Fred (compressed)	002		Fred Smith	76543
000000	00000	XYZ Company (compressed)	001		XYZ Company	57692
057692	00001	23 Acacia Avenue (compressed)	001		23 Acacia Avenue, Newtown	
057692	00002	01234123456	001	D	0 1234 123456	
057692	00002	03123456821	002		0 3123 456821	
056792	00003		001		GBP 10000	
056792	00004		001		retail	
065737	00001	74 Station Road (compressed)	001		74 Station Road, Oldtown	
065737	00002	02378665443	001		0 2378 665443	
065737	00002	07123456821	002		0 7123 456821	
065737	00003		001		GBP 15000	
065737	00004		001		retail	
076543	00001	48 Mill Road (compressed)	001		48 Mill Road, Other Town	
076543	00003		001		GBP 000	
076543	00004		001		retail	

Fig. 5

Note that there are no problems with variable amounts of data for different customers. There do need to be mechanisms that ensure that some or all attributes are mandatory and to maintain data integrity but these are not described here.

The name sequencing mechanism allows us to access customers by name instantly, without database searching. But what if we do know the customer number and want to access a record by customer number? If we add a record sequence field as field 3, and as part of the key field, with, say, 0 for records in name sequence and 1 for records in number sequence, we can store extra records in customer number sequence and display and retrieve the records in customer number sequence or by individual customer number.

KEY FIELD					CONTROL DATA	USER DATA		
Record Number	Attribute Number	Recd Sequ	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number	Link Field
000000	00000	0	Smith, Fred (compressed)	001		Fred Smith	65737	
000000	00000	0	Smith, Fred (compressed)	002		Fred Smith	76543	
000000	00000	0	XYZ Company (compressed)	001		XYZ Company	57692	
000000	00000	1	57692	001		XYZ Company	57692	
000000	00000	1	65737	001		Fred Smith	65737	
000000	00000	1	76543	001		XYZ Company	76543	
057692	00001	0	23 Acacia Avenue (compressed)	001		23 Acacia Avenue, Newtown		
057692	00002	0	01234123456	001	D	0 1234 123456		
057692	00002	0	03123456821	001		0 3123 456821		
056792	00003	0		001		GBP 10000		
056792	00004	0		001		retail		
065737	00001	0	74 Station Road (compressed)	001		74 Station Road, Oldtown		
065737	00002	0	02378665443	001		0 2378 665443		
065737	00002	0	07123456821	002		0 7123 456821		
065737	00003	0		001		GBP 15000		
065737	00004	0		001		retail		
076543	00001	0	48 Mill Road (compressed)	001		48 Mill Road, Other Town		
076543	00003	0		001		GBP 000		
076543	00004	0		001		retail		

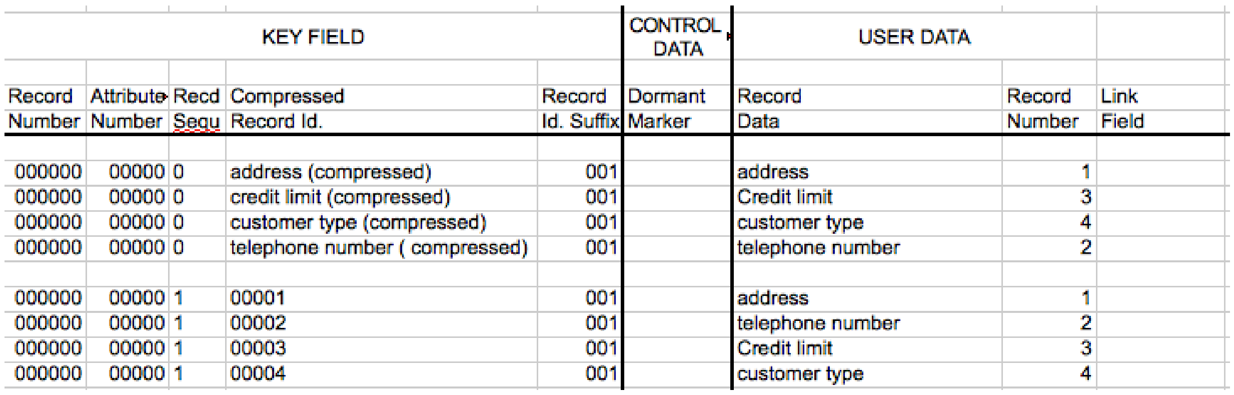
Fig. 6

The link field is used to ensure that the two records for each customer name are linked and that one cannot exist without the other. It is not seen by users.

We can now access customer records in either name or number sequence without the need to sort them.

We might not want to retrieve all attribute values for a customer – perhaps we only wish to find their telephone number. As the file is indexed, having accessed the customer name, we only then need to access the telephone number record(s). We could define in a program that the telephone number is attribute 2. However there is a simple way of achieving this without programming.

Having established the database mechanism, we can use it for accessing other record types in other files or tables by name and/or number. We can create a table of attribute names, which, as they are created, are automatically allocated attribute numbers. We can then see all attributes in name sequence together with their numbers.



KEY FIELD					CONTROL DATA	USER DATA		
Record Number	Attribute Number	Recd Sequ	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number	Link Field
000000	00000	0	address (compressed)	001		address	1	
000000	00000	0	credit limit (compressed)	001		Credit limit	3	
000000	00000	0	customer type (compressed)	001		customer type	4	
000000	00000	0	telephone number (compressed)	001		telephone number	2	
000000	00000	1	00001	001		address	1	
000000	00000	1	00002	001		telephone number	2	
000000	00000	1	00003	001		Credit limit	3	
000000	00000	1	00004	001		customer type	4	

Fig. 7

We can give the attributes proper names, such as telephone number, rather than, say, TELNUM. The same database handler mechanism would automatically allocate numbers to each new attribute as it was added. Thus, in the earlier example, address would be 00001, telephone number would be 00002, etc. A list of all attributes could be displayed in alphabetic sequence of attribute name, together with the attribute number.

This table of attribute names has the same layout as the Customer file and shows that there is very little difference between user data, such as customer details, and meta data, such as attribute definitions. The attributes are now defined in the database rather than in program code. As new attributes can be defined with ease, file maintenance is no longer a significant problem. When new attributes are defined for a file, the existing data does not need to be changed. From the point of view of an application developer, this table of attributes can be considered as his user data. Only when it is used in a customer application does it become meta data.

The database handler can be set up so that, when a customer record has been accessed, the operator is presented with a list of attributes, stored in a file, from which he or she can choose the one(s) required. The attribute number will be retrieved and the key field of the required attribute data record is now known and the record can be retrieved. If there are many attributes, the user can either page through the list or type in the full (e.g. telephone number) or generic (tel*) name of the attribute required.

Alternatively, the database handler can be set up so that all or a subset of attributes are retrieved. These can be displayed in a sequence selected by the application developer and they can be displayed as a single contiguous record.

We need a mechanism that allows us to access individual attribute values for a customer so that if an application only required access to, say, customer name and telephone number, then, as the file is indexed, only those two records need to be accessed. The saving in processing time and disk reads may not be so significant with the simple example above, but, in a real world situation, there would be many more fields in a customer file, and this file structure and access method considerably reduces response times.

Such lists of attributes or meta data could be stored in menus. To distinguish between user data and meta data, such as menus, we can include an additional field, Record Type, as the first field in the key.

The following table shows how, having accessed a customer name record, the database handler will present him with a list of the attributes available from which he can select the one(s) that he requires.

KEY FIELD						CONTROL DATA	USER DATA		
Recd Type	Record Number	Attribute Number	Recd Sequ	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number	Link Field
0	000000	00000	0	Smith, Fred (compressed)	001		Fred Smith	65737	
0	000000	00000	0	Smith, Fred (compressed)	002		Fred Smith	76543	
0	000000	00000	0	XYZ Company (compressed)	001		XYZ Company	57692	
User selects XYZ Company and is then presented with a list of available attributes about the XYZ Company from which he can select (or request all). Although very similar in structure to user data, the menus are stored in a separate file from the user data.									
1	000000	00000	0	address (compressed)	001		address	1	
1	000000	00000	0	credit limit (compressed)	001		Credit limit	3	
1	000000	00000	0	customer type (compressed)	001		customer type	4	
1	000000	00000	0	telephone number (compressed)	001		telephone number	2	
If he selected telephone number, the database handler would only retrieve the following record. The user would not see the deleted telephone number unless he had the necessary authority to look for deleted numbers.									
0	057692	00002	0	03123456821	001		0 3123 456821		

Fig. 8

Many entity types and attributes are already defined in the ERROS Business Model. Developers can add additional attributes to existing entity types or they can create their own entity types, and use attributes already defined within ERROS or add new ones using the terminology chosen by their users.

Because the attributes can be accessed by name, a user viewing the customer table could type

xyz company/tel*/*

and the database would immediately display the telephone number of the XYZ Company. No SQL is required.

Using the same mechanism, we can create tables in the database to define entity types, applications, users, etc.

If a further field, entity type (i.e. file) number is put at the beginning of each record, and is indexed as part of the key field, then the same basic database structure can be used for all data and all data for all entity types might be stored in one single file. Only one database handler is required and this can be used for both data and application definition and for operating the applications created. Here is the same data with the Entity Type No.

	KEY FIELD						CONTROL DATA	USER DATA		
Entity Type No.	Recd Type	Record Number	Attribute Number	Recd Sequ	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number	Link Field
12345	0	000000	00000	0	Smith, Fred (compressed)	001		Fred Smith	65737	
12345	0	000000	00000	0	Smith, Fred (compressed)	002		Fred Smith	76543	
12345	0	000000	00000	0	XYZ Company (compressed)	001		XYZ Company	57692	
12345										
12345	0	000000	00000	1	57692	001		XYZ Company	57692	
12345	0	000000	00000	1	65737	001		Fred Smith	65737	
12345	0	000000	00000	1	76543	001		XYZ Company	76543	
12345										
12345	0	057692	00001	0	23 Acacia Avenue (compressed)	001		23 Acacia Avenue, Newtown		
12345	0	057692	00002	0	01234123456	001	D	0 1234 123456		
12345	0	057692	00002	0	03123456821	001		0 3123 456821		
12345	0	056792	00003	0		001		GBP 10000		
12345	0	056792	00004	0		001		retail		
12345										
12345	0	065737	00001	0	74 Station Road (compressed)	001		74 Station Road, Oldtown		
12345	0	065737	00002	0	02378665443	001		0 2378 665443		
12345	0	065737	00002	0	07123456821	002		0 7123 456821		
12345	0	065737	00003	0		001		GBP 15000		
12345	0	065737	00004	0		001		retail		
12345										
12345	0	076543	00001	0	48 Mill Road (compressed)	001		48 Mill Road, Other Town		
12345	0	076543	00003	0		001		GBP 000		
12345	0	076543	00004	0		001		retail		

Fig. 9

Although the whole database, both user data and meta data, could now be accommodated in one single file, there are operational reasons for splitting it into just a few files (under 10). These all have identical layouts and are all accessed and updated by the same database handler. As the files are opened when the operator signs on, remain open until the operator signs off, are not constantly being opened and closed, and no new database programs are being called, performance is outstanding. ERROS provides high level security at the attribute level.

The menu records containing attribute lists currently contain all attributes defined for an entity type. If, when an operator signs on, he or she selects an application to which he is authorised, the database handler will retrieve the application number. If this is used in the menu tables as the Application or record number, then application dependent menus can be created as in the following simple examples.

If the entity types are also defined in a table, then the records can be accessed by name or number, as before. So a user could type

entity type/customer/xyz company/tel*/*

and the database would immediately retrieve the telephone number of the XYZ Company. The ERROS Connectionist Database can be described as intelligent. Again, no SQL is required. To create a new table, the application developer simply has to add its name to the table of entity types. The database handler will allocate an entity type number.

	KEY FIELD						CONTROL DATA	USER DATA		
Entity Type No	Recd Type	Application Number	Attribute Number	Recd Sequ	Compressed Record Id.	Record Id. Suffix	Dormant Marker	Record Data	Record Number	Link Field
12345	1	00032	00000	0	address (compressed)	001		address	1	
12345	1	00032	00000	0	credit limit (compressed)	001		Credit limit	3	
12345	1	00032	00000	0	customer type (compressed)	001		customer type	4	
12345	1	00032	00000	0	telephone number (compressed)	001		telephone number	2	
12345	1	00321	00000	0	address (compressed)	001		address	1	
12345	1	00321	00000	0	telephone number (compressed)	001		telephone number	2	

Fig. 10

This shows how different versions of the same customer record can be displayed in different applications.

Attribute definition records contain multiple fields and these determine how the data records within each attribute are to be identified and accessed and then displayed by the presentation layer. They also specify the security levels required by operators to add, change, delete and many other functions for the attribute. The values in individual menu records for the same attribute might be different, so that, for instance, in application 32 above, addresses might be displayed on one line, immediately followed on the same line by the telephone number, perhaps with a comma as a separator, whereas, in application 321, the address might be displayed on several lines, with each telephone number on the lines below. Security levels might be different – perhaps, in application 321, the user might have read only access and cannot change addresses or telephone numbers unless he can operate application 32.

The data in each attribute can be processed and presented by an exit program. One exit program may be suitable for a group of different attribute types. This might be a standard ERROS exit program or a special user version. User versions of menus can also be created. These two most powerful features allow extraordinary flexibility and enable the creation of user specific versions of standard ERROS applications without the need to alter those. There can be a mix of standard and user menus. Exit programs can also be used to read and write non-ERROS data and pass these to and from ERROS and also carry out any non-ERROS functions.

This document outlines the basic principles for building a new type of database that can be defined and created without physical database design, that allows incremental application development without programming and that can "grow as you go". The way in which ERROS actually works is different in some respects from this theoretical outline. For instance, the ERROS key field contains more sub-fields than shown in the figures above.

There are many further details that need to be considered to achieve a full understanding of ERROS and its extraordinary power and flexibility. Because the ERROS Connectionist Database is in fact a semantic network and access to each node is provided by an attribute definition, it is possible to define in the database how the data records are to be processed at the attribute level, and there can be multiple definitions for any attribute, depending on the application and the route taken to the data. With traditional databases, where a single record contains multiple fields, program coding or perhaps SQL will be necessary to manipulate the different attribute values. As ERROS is a semantic network, all data is stored in its context and it is a contextual database.

ERROS is able to store very complex multi-dimensional, many-to-many relationships. Relationships can be created with individual records at the attribute level. Subject to their security, users can navigate all relationships in either direction at the same very high speed, without the joins of relational databases, which do not naturally store bi-directional or many-to-many relationships. Modelling complex relationships and hierarchies using relational databases is not easy yet with ERROS it is very straightforward. The relational features of ERROS, which were used to construct ERROS itself, need to be considered separately.

With traditional relational databases, response times are very dependent on the design created by the developer, whereas, with ERROS, they are consistently very rapid, whatever the database function being performed and whatever the size of the database. Tests have suggested that ERROS response times can be as little as one tenth of those achieved with a traditionally created application using a relational database.

ERROS application development and the applications created are all object-oriented as encapsulation, multiple inheritance and polymorphism are all built into the basic logic of ERROS and included in every ERROS application by default. ERROS application developers do not need any understanding of OO concepts.

Application development and maintenance is much more productive using ERROS than with any other method, with gains of several hundred percent. In one test, it had taken three days just to design a simple traditional application and a total of about two weeks to complete its creation. The whole process took just three hours using ERROS. The design was not required.

The same database handler and main ERROS programs are used for both application creation and for the operation of those applications. Only three main ERROS programs and one exit program are required and these total just under 17Mb of compiled code, as a large part of the ERROS functionality is defined in the ERROS database using relationships. In other words, as already stated, ERROS was used to define itself.

Most of the time, no new files, programs or other objects are created during application creation, as data, application, and security definitions are all stored in the integrated ERROS database together with the user data. All changes to data and application definitions and to user data are journalled and there is an audit trail detailing who changed every single piece of meta and user data and when. All ERROS applications are automatically integrated, sharing data as appropriate. An ERROS high availability option is easily implemented.

The sample data illustrated above are very simple, but ERROS can be used to build very complex databases for most commercial purposes and for the humanities. It is very suitable for the creation of a wide range of robust, secure, high performance, high availability applications, including transaction processing. It is equally appropriate for the creation of databases where the amount of data can vary dramatically from one record to another, such as in the humanities where, in many records, only a small percentage of attributes may contain data whilst in others there may be very many attribute iterations. This is true, for instance, for recording history where data for one event, person, concept, etc., may come in small droplets and then, suddenly, and without warning, in a flood. These features are only possible because of the vertical nature of the database. ERROS applications turn raw data into a fully navigable semantic knowledge base.

ERROS is totally unlike traditional database and application development tools. There are no compiled ERROS applications - the concept of compiled applications has gone. ERROS creates HTML and Javascript on the fly so that all applications, including data definition and application creation, can be operated immediately over the internet using the ERROS Standard Operating Interface. For printing, ERROS generates PCL/5 on the fly. Report layouts are defined in the database. This functionality is included in the ERROS programs mentioned earlier.

An important feature of ERROS is that any entity can belong to multiple entity types and inherit the attributes of the entity type under which it is being considered. ERROS has no need for foreign keys or joins and can handle views, triggers and stored procedures.

ERROS is also a real-time business modelling tool and a prototyping tool. The prototypes created are working systems, not simulations. The database and application definitions can be extended dynamically. No longer do the user requirements and system specification have to be defined in every particular before detailed database design can be begin. Not only is there no need for physical database design in ERROS, there is no facility for this as the database layout can be extended incrementally as new needs arise, without the need to change existing database records.

Very large traditionally created applications may be too complex for any single person to be able to comprehend fully. With ERROS, the database and applications are created in small steps that can be readily understood. These are automatically integrated by ERROS and, if the individual definitions are correct, then the whole should be error-free, particularly if, as will often be the case, no new programs have been created and even though its complexity may defy overall understanding.

22nd June 2017

Rob Dixon

rob.dixon@erros.co.uk