Surviving Client/Server: Database Design Primer

by Steve Troxell

Designing tables for a client/server database can present new challenges to traditional file server oriented developers. There are many more features and issues in the database server world that the developer should to be aware of. Nearly any reference book will tell you what the additional features are, but you also need to now how to employ them effectively.

I get some of my best pearls of wisdom from pop culture. About ten years ago I saw a comedy skit that pretty much summed up my opinion of a lot of people in software development. The skit involved a quasi-prehistoric tribe of people out on a hunt, trying to formulate a plan for felling some animal, a buffalo let's say. The leader suggested a head-on assault and commented that 3 or 4 of their party would probably be killed, but they would have food for the tribe for a whole month. A younger member of the hunting party came up with an idea. "What if we all spread out and waved our arms and shouted and drove the animal over that cliff over there? Then we can simply walk down to the base of the cliff and drag the carcass back to camp." The leader of the hunting party thought for a while and said, "A very interesting plan. But your idea is new and we fear new things. Therefore it must be rejected."

People entering client/server development are faced with a variety of new capabilities as well. It surprises me to find that some developers still shy away from such things apparently simply because they are unfamiliar with them. The following is meant to shed some light on some of the issues facing a client/server developer.

Open Architecture

The most important issue you must come to grips with as early as possible is whether your system will be an open or closed architecture. Nearly all client/server RDBM systems are designed to allow an open architecture, but that doesn't mean you must have one. What I mean by open architecture is the database can be accessed not only by the proprietary applications you develop, but also by off the shelf third party productivity tools such as spreadsheets, data extractors, word processors and report generators. This is generally achieved by using a database driver such as ODBC to connect the application to the database server. Third-party applications can be written to conform to the standard ODBC interface and the ODBC driver in turn is written to access a particular database server's API. In this way the same application can access different vendor's databases by substituting an ODBC driver appropriate for that database.

Many file server databases, such as Paradox or dBase, provide for open architecture as well, so this isn't the exclusive domain of client/server. However, most developers are accustomed to closed architectures where they are not concerned about anything but strict proprietary application access to the database. It's important that developers and customers both understand the role of third party software in the system as early as possible, because many aspects of the database design can be an aid or hindrance to an open architecture.

If the extent of your system's need to provide external access to third-party programs is simply allowing your users to run prefabricated reports through a commercial report writer, then you really don't have much of an open architecture to worry about. While you'll still need to consider some issues of user permissions to tables and other objects, your users have no need to know what the tables are, their layout, what the fields are, the relationships between tables, etc. But for a mere technicality, this is still essentially a closed system. But if your users will be designing their own ad hoc reports or queries, then you'll need to consider an open architecture.

The real value of an open architecture is allowing at least some cross section of the user base direct access to at least some of the data beyond a prefabricated framework. Generally, this access is provided in the form of ad hoc querying for data analysis and reporting. This is a very valuable quality for a database to have as it opens up selected bits of data to more users so they can get what they need and how they need it in a timely manner. This means you may make different database design choices than you would if the design was the sole domain of software developers.

User Security

Open architectures are the principal reason why all client/server databases provide user login accounts. A typical closed database system might not be concerned with exactly who was doing what in the application. Or if it was, it might employ a simple `Users` table within itself. Some true client/server applications that are unconcerned with user accountability and external access to data simply create a single fixed account in the database which is used by all applications to gain access to the data. The account username and password are encoded directly into the applications and there is no login dialog for the user to complete.

However, user accounts in client/server databases are not primarily concerned with tracking who's connected and who's not. Their chief purpose is in controlling who has access to which parts of the database. Remember, third party tools connect to the database independently, and the whole database is laid bare for all to see. Would you want marketing people, whilst creating ad hoc reports on product sales, to see the payroll data for the whole company? On the other hand, you do want the accounting personnel to be able to get to the payroll data? When all access is running through proprietary applications, you can easily control who gets to what. But with an open architecture, you have to rely on the built-in security available via the user accounts in the RDBMS you're using.

This means granting or revoking permissions to individual users to read, insert, update or delete data (or any combination) for a given table, as shown in Listing 1. So, no permissions would be granted on payroll tables to any user that wasn't in the accounting department (see Listing 2). Many RDBM systems let you create groups of users in just this manner to make it easier to manage user permissions.

Actionable Information

Database design involves striking a balance between performance, functionality, and size. With an open architecture there is also the need to optimize the amount of "actionable information" within the data. This means data that carries meaning in and of itself, minimizing the requirement to decode, translate or interpret the data. Software developers are accustomed to decoding, translating and interpreting data and design their databases accordingly, to achieve compactness and because there's been no compelling reason not to. With an open architecture you have to keep in mind that users other than software developers will be working with the data.

Take a look at the data shown in Figure 1. Which rows are for open orders? Which rows are for orders on hold? Many database developers don't think twice about using a numeric sequence to encode values for a column. But the values carry no meaning in and of themselves. You are forced to make a translation from a completely abstract value by referring to a data encoding sheet or joining to a lookup table.

Now look at Figure 2. The same data has been encoded differently. Sure, you can say a mental translation of the code value `OP` to the term "open order" is still required, but the value is now a mnemonic rather than an abstract value. It carries meaning by itself.

To get the same level of meaning in a result set from the abstract data shown in Figure 1 would require a join to a lookup table. Several columns employing abstract encoding could lead to several joined tables and you can see how quickly ad hoc querying can become complicated and inefficient.

Declarative Constraints

If your open architecture allows data modifications from third-party programs, then you'll definitely want to consider using some form of automatic processing to perform data validation and ensure data integrity. With client/server databases, you have two choices: declarative constraints or triggers. Declarative constraints are defined at the time you create the table. They include nullability, defaults, uniqueness and check constraints. Listing 3 shows a few examples. The exact syntax varies from vendor to vendor so be sure to check your manual before trying this with your database server.

Any field may contain a null value. A null indicates the actual value is not available or not known. For example, if an employee does not wish to divulge their birthday, we may store a null in the `DateOfBirth` field. Nulls are available to all datatypes and save the developer from having to contrive a special

➤ *Listing 1*

```
/* Give John read-only access to the employees table */
GRANT SELECT ON Employees TO JohnA
```

➤ *Listing 2*

```
/*Deny all access to the payroll table */
REVOKE ALL ON Payroll FROM Public
GRANT ALL ON Payroll TO Accounting
```

➤ *Figure 1*

OrderNum	CustNo	Date	Status	Clerk	Total
871182	113	6/2/97	0	78	$507.92
871183	291	6/2/97	0	78	$122.45
871184	88	6/2/97	1	54	$1,209.00
871185	195	6/2/97	3	99	$45.22

➤ *Figure 2*

OrderNum	CustNo	Date	Status	Clerk	Total
871182	113	6/2/97	OP	78	$507.92
871183	291	6/2/97	OP	78	$122.45
871184	88	6/2/97	HD	54	$1,209.00
871185	195	6/2/97	VD	99	$45.22

code value to serve the same purpose. However, nulls introduce some complexities to query logic. You can disallow nulls in any column by using the NOT NULL constraint. We will explore nulls more closely later in the article.

When a column value is omitted from an SQL INSERT statement, then the values for that column in the rows being inserted will be null by default unless you supply your own using the DEFAULT constraint. If you've declared a column NOT NULL, then you should supply a default value unless there is no logical default (like the DeptNo field in the example). Defaults can be literal values or functions built into the database. The HireDate field in this example uses the hypothetical built in function CurrentDate to post today's date as a default value.

Some columns within a table require non duplicating values, like employee badge numbers. We can enforce this rule by using the UNIQUE constraint, which simply places a unique index on the column. Databases vary in how they handle null values on unique columns. Some require that there be no more than one row with a null value in the column (strict interpretation of uniqueness). Others allow any number of rows to have a null value in the unique column, following the rule that no null value can be equal to any other null value (strict interpretation of null).

In the case of our badge number column, our business rule is that not every employee is required to have a badge number, but those who do cannot have duplicate badge numbers. The only way we can enforce this with declarative constraints is to define the column unique and allow any number of null values within the unique column. If we chose to use a special non-null code value for employees without badges, we could not use the unique constraint.

Check constraints are the most flexible declarative constraints. Check constraints allow you to define an expression that must evaluate to true. In Listing 3 we've used check constraints to ensure that the Salary field is a positive number and that the BadgeNo field is a null or a string of one uppercase letter followed by two digits.

Nearly all declarative constraints can be given a name, like our check constraint for the Salary field. When a constraint is violated by an insert or update operation, the server raises an error and the name of the constraint being violated should be present in the error message text. In this way, client apps can respond to particular errors, parse the error text for constraint names, and provide a cleaner, more meaningful error message to the user.

➤ *Listing 3*

```
CREATE TABLE Employees(
  EmpNo        integer       NOT NULL,
  FirstName    varchar(15)   NOT NULL DEFAULT '',
  LastName     varchar(20)   NOT NULL DEFAULT '',
  DeptNo       char(3)       NOT NULL,
  PhoneExt     char(3)       NOT NULL DEFAULT '',
  HireDate     date          NOT NULL DEFAULT CurrentDate,
  DateOfBirth  date          NULL,
  SSN          char(11)      NOT NULL DEFAULT '<unknown>',
  Salary       double        NOT NULL DEFAULT 0
                               CONSTRAINT SalaryPositive
                               CHECK (Salary >= 0),
  BadgeNo      char(3)       NULL UNIQUE
                               CHECK (BadgeNo IS NULL or
                                      BadgeNo LIKE '[A-Z][0-9][0-9]')
);
```

➤ *Listing 4*

```
CREATE TRIGGER BadgeRule ON Employees
FOR INSERT, UPDATE
AS
BEGIN
  IF (New.DeptNo IN ('012', '014', '155')) AND
     (New.BadgeNo IS NULL) THEN
    EXCEPTION BadgeNoReqd;
END
```

Triggers

Declarative constraints only go so far. For example, how do we enforce the rule that employees in departments 012, 014 and 155 require badges? For more powerful automatic data processing, you have to turn to triggers. Triggers contain static SQL code that fires whenever a data modification operation is performed, regardless of the application that initiated the operation. Within a trigger you can generally use more complex logic than would be available in a declarative constraint. Separate triggers can be placed specifically for insert, update or delete operations on a given table. Some databases allow many triggers to exist for the same operation on the table, each firing in sequence.

Listing 4 shows a trigger we might use to enforce the BadgeNo rule. Again, exact syntax varies from database to database. Triggers provide a mechanism for examining the values of the row being affected. In this case, the New variable gives us access to the values being written to the table.

A common use of triggers is automatic stamping of username, date or time information in sensitive tables. Also, cascading deletes can be implemented with triggers, where deleting a record in a master table automatically deletes associated records in one or more detail tables (see Listing 5).

You'll want to be judicious in your use of triggers and constraints since there is a bit of overhead in their execution, and they will always execute for every data modification they are associated with.

Domains

Domains are essentially user-defined datatypes in SQL. Domains are very helpful for ensuring the

consistency of data declarations across tables in a non-trivial database. For example, people's names might appear in several different tables: customer contacts, salespeople, employees, vendor contacts, etc. If a design decision has been made that all name fields allow up to 30 characters, then it is much simpler to define a `TPersonName` domain as `VARCHAR(30)` and make all your table column definitions with the domain name rather than the direct data type as shown in Listing 6.

Domains can be valuable for keeping consistent definitions and making it much easier to change definitions across the entire database. A frequently overlooked value of domains is their use in stored procedure parameter datatypes. While there may be only a few fields in table definitions sharing the same domain, there may be many different stored procedures with parameters of that same data type. Once again, ensuring the consistency of the data type definitions within stored procedure parameters and their associated table columns is much easier with a domain name.

Normalization

Normalization generally refers to eliminating redundancies in tables by splitting them apart into smaller tables with a relationship defined between them. Typically several narrow (fewer columns) tables are better than one wide (lots of columns) table. The narrow tables usually result in more compact data because omissions and variable quantities can be handled more efficiently. Also, a narrow table means more rows are packed into a data page and the server can scan more rows per I/O operation, improving performance. There are five main rules of normalization. In practice the first three are usually sufficient for most systems.

First Normal Form: Atomic Values

The point where a column and row intersect in a table we'll call a cell. First normal form requires that each cell in a table contain an atomic value. That is, a single value cannot be further broken down into useful values. For example, for a table of mailing addresses, it is not uncommon for designers to group the elements city, state and postal code into a single field of the form `<city>, <state> <postal code>` since that is how they need to displayed when printed on a mailing label. However, this single value can be broken down into three useful elements and should be stored in three separate columns. The values in the three columns can easily be combined to produce the desired format for the mailing label.

Having the values separated out permits you to manipulate the data at a finer level. For example, most bulk mailings can gain a postage discount if the addresses are sorted by postal code. How are you going to sort the records when all three elements are pushed together in a single field. Also you may want to stagger mailings to different states. You will need to select all addresses for state A for the first mailing and all addresses for state B for the second mailing. It would be extremely difficult to filter records on state if it's not broken out into its own field.

You shouldn't get carried away with atomizing values though. For example, given a phone number with area code, you could argue that the area code must be separated from the actual phone number to be in first normal form. Generally it is not necessary to go to this extreme if the phone number and area code will always be used together, which is nearly always the case. You should only be concerned about this if your system has some functional need to isolate the area code, for example a requirement to produce a list of contacts sorted by area code.

This reasoning also applies to people's names. Many designers create a single field for a name, such as `Steve Troxell`. If there is absolutely no requirement to operate on the individual elements of the name, this may be fine. For example, a contact name for a customer or vendor record.

However, there usually is much value in splitting name fields into first name and last name components that may not be immediately obvious. Is it important to you that the appearance of the name be consistent? If you have a single name field how are you going to prevent problems like user A always enters `Steve Troxell` but user B always enters `Troxell, Steve`. If you sort by the name are you making an assumption that all data entry will be `lastname, firstname`? If there is the possibility that these issues may arise in your system, it's better to split the name field apart; you can always assemble the complete name in any form you need.

First Normal Form: No Repeating Groups

An extension of the requirement of atomic values is that there be no repeating groups of information. That is, a single column cannot be an array, list, or multi-field structure. Most RDBM systems enforce this by simply not providing

➤ *Listing 5*

```
CREATE TRIGGER MasterCascade ON MasterTable
FOR DELETE
AS BEGIN
  DELETE FROM DetailTable WHERE KeyNo = Old.KeyNo;
END
```

➤ *Listing 6*

```
CREATE DOMAIN TPersonName AS varchar(30);
CREATE TABLE Customers(
  CustNo          integer;
  Company         varchar(30);
  Contact         TPersonName);
```

non-scalar datatypes. However, developers still try to buck the system by designing tables like that shown in Figure 3 .

This is in first normal form by technicality only. Each field is an atomic value. But the layout still breaks the spirit of first normal form and should be redesigned. The `AuthorID` fields presumably link to a separate `Authors` table containing each author's name and other info. This approach introduces several problems. First, it imposes an artificial business rule that there can be no more than three authors for any given book. So do we simply refuse to accept a book with four or more authors? Most likely, the book will get entered but only the first three authors will be credited. Second, if I want to see a list of books by any given author, I must query all three `AuthorId` fields to guarantee coverage of all possible positions of the value I'm looking for. This can lead to some hideous code workarounds to effectively implement a filter on author.

A solution to these problems is shown in Figure 4. The `BookAuthors` table contains as many rows for a book as there are authors for that book. You can see how it would be much simpler to search for all occurrences of a single author in the `BookAuthors` table and then link in the book title from `Books`. Tables such as this go by one of several names: relationship, association or junction tables. They describe a relationship between two other tables and don't necessarily contain any useful independent data themselves.

➤ *Figure 3: Books table*

Title	ISBN	PublisherID	AuthorID1	AuthorID2	AuthorID3
Delphi How-To	1-57169-019-0	SAMS	32	45	18
Delphi Unleased	0-672-30499-6	WAITE	16	<null>	<null>

➤ *Figure 4*

Books		
Title	**ISBN**	**PublisherID**
Delphi How-To	1-57169-019-0	SAMS
Delphi Unleased	0-672-30499-6	WAITE

BookAuthors	
ISBN	**AuthorID**
0-672-30499-6	16
1-57169-019-0	18
1-57169-019-0	32
1-57169-019-0	45

➤ *Figure 5*

ProjectID	EmpID	EmpName	BillRate
X1104	102	Bob Smith	65.00
X1104	76	Stan Stevens	35.00
X1256	102	Bob Smith	90.00
X1256	88	Roy Rogers	75.00
J3880	88	Roy Rogers	75.00

➤ *Figure 6*

EmpNo	EmpName	DeptNo	DeptName	DeptManager
112	Sue Bennet	80	Shipping	Bob Smith
115	Roy Robinson	14	Payroll	Carly Johnson
121	John James	80	Shipping	Bob Smith

Second Normal Form

Second normal form means that if a table's primary key is composed of more than one column then no column in the table should be dependent on just part of the primary key. This is easier to see with an example. Figure 5 shows a project table assigning employees to projects. The primary key for the table is both the project identifier and the employee number. As you can see the employee's name is only dependent on part of the key for the row (employee number) and should be eliminated from this table. The `BillRate` column is the hourly rate that employee charges for this project. An employee working on multiple projects might charge different rates for each project. Therefore, the `BillRate` column is dependent on the entire primary key and is appropriate for this table.

The problem with a table that fails to meet second normal form is that if we change a single fact about the database (in this case, an employee's name), then we must change that fact in several different places (in this case, every occurrence in the `ProjEmp` table).

Third Normal Form

Third normal form extends this dependency logic by stating that all non-key columns must depend on the primary key and not depend solely on another non-key column. In Figure 6 it is sufficient to stop with the `DeptNo` column as `DeptName` and `DeptManager` are not directly relevant to the employee, but instead should be broken out into a separate table keyed by `DeptNo`.

There may be occasions where you deliberately break third normal form. Consider the table shown in Figure 7. You might notice that the `Total` column is redundant, being a computation of the `Subtotal`, `Tax` and `Freight` columns. However, if you have an open architecture and users may be querying records based on the order total, it is easier and more reliable to provide the total as a pre-computed value. Or, whether you have an open architecture or not, if there is frequent need to filter orders by total amount (all orders over $1000, for example), providing an explicit column for the value allows you to index it for faster retrieval. If you elect to do this, you may want to consider having triggers on the table to calculate the `Total` column.

OrderNo	Subtotal	Tax	Freight	Total
80221	114.50	6.87	10.00	131.37
80222	505.75	30.35	25.00	561.10

➤ *Figure 7*

Null Values

There is perhaps no greater controversy in relational databases than the use of null values. E F Codd, widely regarded as the "father" of relational databases, decreed that a proper relational database system have something like null to represent missing or inapplicable data. In fact, Codd wanted *two* values: one for incomplete data to be filled in later and one for inapplicable data that would never be filled in. On the other hand, C J Date, a leading expert on relational databases, insists that nulls are evil and should be done away with.

Nulls do create problems, but like most things, they serve a purpose. Nulls shouldn't be shunned outright "because they are new and we fear new things." Nulls can cause more problems than they solve, so become familiar with their uses and pitfalls and use them appropriately.

Nulls create confusion when users fail to account for them. For example, a user might select all employees with a salary greater than 50,000 and find 10 qualifying rows. He might then run a query of all employees with salary less than or equal to 50,000 and find 17 qualifying rows. He might come to the conclusion that there were 27 total employees in the company. This would be incorrect if there were some employees with null salaries (hourly wage employees perhaps).

How nulls sort varies from vendor to vendor, but they sort out either before all other values or after all other values. This implies that nulls are either less than or greater than all other values and may be misleading to users.

A null present in any term of a mathematical expression results in the entire expression evaluating to null. If any part of the expression is unknown, how can any legitimate result be obtained? However, with a special value like zero instead of null, you'd get a valid result but it would be wholly inaccurate. Aggregate functions in SQL like `AVG` and `SUM` skip null values for their computations and produce a result that includes all the non-null values. In this respect, nulls might be helpful because you can obtain values for the data that is known. If you had used special codes instead of nulls, `AVG` and `SUM` would be useless.

Going back to our example in Listing 3, how would you represent a missing birthday in a date field? The system would most likely reject an invalid date like 99/99/99, so you'd have to contrive an unlikely though legitimate date, like 1/1/2100, to represent no date.

Although most problems with nulls are the result of users or developers failing to account for them, they are best avoided unless serving some purpose. You should define all column tables as `NOT NULL` and only allow nulls in specific cases that improve your design.

Conclusion

Client/server systems offer more facilities to have the database itself, rather than client applications, aid in managing data. Developers new to client/server may be unaware or unfamiliar with these features and hopefully this brief introduction entices you to explore further. Like many things, features should not be embraced or shunned by rote rules alone. It's much better to understand what each can do and what possible problems may arise in its use. Then select the appropriate tool for the problem at hand.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@turbopower.com or on CompuServe at 74071,2207