CHAPTER 25

Data Quality

Abstract

This chapter covers issues associated with data quality. It begins with a discussion of the importance of data quality (being able to trust the accuracy of data in a database). The remainder of the chapter covers recognizing and handling specific types of data quality problems. The last section looks at the relationship between an organization's employees and data quality.

Keywords

data quality
complete data
incorrect data
incomprehensible data
inconsistent data

As early as the 1960s, there was an expression in computing that most people in the field agreed was true: "Garbage in, garbage out." (It was abbreviated GIGO and was pronounced "guy-go.") We worried about the effect of the quality of input data on the output of our programs. In the intervening years, during the rise of databases, GIGO was largely forgotten. Today, however, with some data warehouses exceeding a petabyte of data, the quality of that data has become extremely important once again.

Exactly what do we mean by data quality? To be useful, the data in a database must be accurate, timely, and available when needed. Data quality assures the accuracy and timeliness of data and, as you will see, it is much easier to ensure data quality *before* data get into a database than once they are stored.

Note: Much of what we do to ensure data quality is just plain common sense. In fact, you may find yourself saying "well, that's obvious" or "of course you would do that" as you read. The truth is, however, that as logical as many data quality procedures seem to be, some organizations simply overlook them.

Why Data Quality Matters

Why do we care so much about data quality? Because we need to be able to trust that what we retrieve from a database is reliable. We will be making both operational and strategic decisions based on what we retrieve from a database. The quality of those decisions is directly related to the quality of the data that underlies them.

Consider, for example, the decisions that a buyer makes for the clothing department of a retail chain such as SmartMart. Choices of what to stock for the winter holiday shopping season are made nine to 12 months in advance, based on what sold the previous year and the buyer's knowledge of clothing styles. The buyer therefore queries the operational database to create a report showing how much of each style sold at each store and on the Web. She can see that jeans sell well in one particularly store in a region, while another store sells more dress pants. She will then adjust her orders to reflect those sale patterns.

However, if the sales data are incorrect, she runs the risk of ordering the wrong type of merchandise for each store. SmartMart can certainly move inventory from one store another, but during the holiday shopping season, customers are often unable to wait for merchandise to arrive. They will either purchase a different item or go to another retailer. In the long run, SmartMart will lose sales.

We can probably come up with hundreds of scenarios in which inaccurate data cause problems for businesses: customers who can't be contacted because of out-of-date phone numbers or e-mail addresses, orders missing items that are never shipped, customers who aren't notified of recalls, and so on. The bottom line is that when we have problem data, business suffers.

Note: It is often better to have a database application crash than it is to have a report that contains inaccurate results. In the former case, it's clear that you have a problem; in the latter case, there may be no indication that the report is invalid, so you'll go ahead and use it, bad data and all.

Data quality problems arise from a wide range of sources, and have many remedies. Throughout the rest of this chapter we will look at variety of data quality ills, how they are likely to occur and what you can do to prevent them, or at least minimize their occurrence.

Recognizing and Handling Incomplete Data

One source of data quality problems is missing data. There are two general sources: data that are never entered into the database and data that are entered but deleted when they should not be.

Missing Rows

Missing rows occur for a number of reasons. One common reason is that someone has been using low-level data manipulation tools to "maintain" the database and, in the process, either deleted rows or missed rows when copying data from one table to another.

the database and, in the process, either deleted rows or missed rows when copying data from one table to another.

Note: The ability to manipulate rows in this way with a low-level tool violates one of Codd's rules for a relational database, which states that it should not be possible to circumvent integrity rules with low-level tools.

Missing rows can be very hard to detect if their absence doesn't violate referential integrity constraints. For example, it's impossible to detect that an item is missing from an order—until the customer contacts you about not receiving the item. It's not necessarily obvious, however, where the error has occurred. Any of the following might have happened:

- An item was lost in transmission from the customer's computer to the vendor's computer (Internet order) or a customer service representative didn't enter the item (phone or mail order).
- An error in one of the vendor's application programs missed entering the row.
- Someone maintaining the database with a low-level data manipulation tool accidentally deleted the item.

It's a relatively easy matter to enter the missing item and ship it to the customer. But what is to prevent the error from occurring again? To do that, we have to find the cause of the error and fix it. Because the possible causes are so diverse, it will be a long, difficult process. Ultimately, we can fix a bug in an application program and put policies in place that control the use of maintenance languages. We can even conduct additional training for data entry personnel to make them more careful with entering order items. We cannot control, however, packets lost on the Internet (or even more problematic, customers who insist that a product was on an order when it wasn't in the first place).

Missing Column Data

As you will recall, SQL is based on what we call three-valued logic. The result of a logical comparison between data and a search criterion can be true, false, or maybe. The maybe occurs when a column contains null because no determination can be made. Sometimes nulls are harmless—we don't care that the data are missing—but in other situations we wish we had the data. Currently, there is no way to distinguish between the two types of null, but some database theorists have proposed that DBMSs should support four-valued logic: true, false, maybe and it's OK, and maybe and it's not OK. It's nulls that fall into the latter category that cause problems with missing data. For example, assume that a customer table has a column for the customer's country. If the customer is in the country where the company is based, a null in the country column doesn't really matter. However, if the customer is in another country, then we need to know the country and leaving that value null leads to problems of undeliverable marketing mailings.

Missing column data can be prevented relatively easily by disallowing nulls in those columns that must have values.

Missing Primary Key Data

A relational database with tight integrity controls prevents data from entering a database because part of a primary key is null. From the viewpoint of data retrieval, this is a positive result because it ensures that every piece of data that makes it into the database is retrievable. However, the lack of a primary key may mean that a set of data are never stored in the database, despite the data being important to the business operating the database.

One situation in which this problem occurs is when entities that are only at the "one" end of relationships have no obvious primary key attribute and no arbitrary key has been created. The primary key is therefore created from the concatenation of two or more columns. For example, if an employee entity has no employee number attribute, then a primary key might be constructed of first name, last name, and phone number. If a phone number isn't available, the data can't be entered because there isn't a complete primary key; the data are never stored. In this case, the solution lies in the database design. Those "top" entities that do not have inherent primary keys should be given arbitrary primary keys. Those arbitrary keys then filter down through the design as parts of foreign keys and ensure that primary key values are available for all instances of all entities.

Recognizing and Handling Incorrect Data

Incorrect data are probably the worst type of problems to detect and prevent. Often, the errors aren't detected until someone external to an organization makes a complaint. Determining how the error occurred is equally difficult because, sometimes, the problems are one-of-a-kind.

Wrong Codes

Relational databases make significant use of coded data. The codes are defined in tables related to the tables where they are used through primary key–foreign key relationships. For example, we often store the names of US states as two-letter abbreviations and then use a table that contains the abbreviation and the full state name for validation and output (when the full state name is needed).

Coding, however, is a two-edged sword. Consider the following scenario: A company divides its products into categories, each of which is represented by a three-letter code. The codes are stored in a table with the code and a description of the product category (for example, PEN and "custom imprinted pens," WCP and "white cups," and so on). When the company decides to carry a new product, the application program used by the staff allows the user to make up a code and enter the code and its description. To make things easier for the user, the user doesn't need to explicitly enter the new code. During data entry, the application captures a failed referential integrity check, automatically asks the user for the code description, and creates the new row in the code table. The problem with this is that, if the user has mistyped a code for an existing product, the application handles it as a new product. Therefore, many codes for the same product could exist in the database. Searches on the correct code will not retrieve rows containing the incorrect codes.

The solution is to restrict the ability to enter new codes. In some organizations, only database administrators have the right to modify the master code tables. If a new product needs to be entered, the DBA assigns and enters the code prior to a clerical worker entering product data. Application programs then retrieve the list of codes and make it available to data entry personnel.

Wrong Calculations

One of the decisions that a database designer makes is whether to include calculated values in a database or to compute them as needed, on the fly. The decision depends on many factors, including the overhead to perform the calculations and how often they are used. We often, for example, compute and store line costs in a line items table:

```
line_items (order_numb, item_numb, quantity_ordered, line_cost)
```

The line cost is computed by retrieving the cost for each item from the *items* table (item number is the foreign key) and then multiplying it by the quantity ordered. The lookup of the price is quick and the computation is simple.

What happens, however, if the formula for computing the line cost is incorrect? Primarily, the total amount of the order (which is also often stored) will be incorrect. It is much more likely that someone will notice that the order total is incorrect, but it may require a bit more investigation to find the specific error and then to track down its source.

Many of the automatic calculations performed by a DBMS when data are modified are performed using *triggers*, small procedures stored in the database whose executions are triggered when specific actions occur (for example, storing a new row). An error in an automatic calculation therefore means examining not only application programs, but all relevant database triggers as well. Nonetheless, once an error in a computation has been identified and corrected, it is unlikely the exact problem will occur again.

Wrong Data Entered into the Database

We humans are prone to make typing mistakes. Despite all our best efforts at integrity constraints, we are still at the mercy of simple typos. If a user types "24 West 325th Street" rather than "325 West 24th Street," you can be sure that the customer won't receive whatever is being sent. The transposition of a pair of letters or digits in a postcode or zip code is all it takes to separate a customer from his or her shipment!

The typographical error is the hardest error to detect because we rarely know about it until a customer complains. The fix is usually easy: edit the data and replace the error with the correct values. However, determining how the error occurred is even tougher than finding the source of missing rows. Was it simply a one-time typing error, in which case an apology usually solves the problem, or is there some underlying application code problem that is causing errors?

The best strategy for separating typos from system errors is to keep logs of errors. Such logs should include the table, row, and column in which the error was detected, when the error was reported, and who was responsible for entering the erroneous data. The intent here is not to blame an employee, but to make it possible to see patterns that may exist in the errors. When multiple errors are in the same column in the same table, the evidence points toward an underlying system problem. When many errors are made by the same employee, the evidence points to an employee who may need more training. However, a random pattern in the errors points to one-of-a-kind typographical errors. (Whether the random errors are at a level that suggests the need for more training for all data entry personnel depends, of course, on the organization and the impact of the errors on the organization.)

Violation of Business Rules

A business often has rules that can be incorporated into a database so that they can be enforced automatically when data are modified. For example, a book club that bills its customers after shipping may place a limit on customer accounts. The orders table includes a trigger that adds the amount of a newly stored order to the total amount owed in the customer table. If the customer table has no constraint to limit the value in the total amount owed column, a customer could easily run up a balance beyond what the company allows.

Yes, it is possible to enforce such constraints through application programs. However, there is no guarantee that all data modification will be made using the application program. In all cases where business rules can be implemented as database constraints, you should do so. This relieves application programmers of the responsibility of enforcing constraints, simplifies application program logic, and ensures that the constraints are always enforced.

Recognizing and Handling Incomprehensible Data

Incomprehensible data are data that we can't understand. Unlike incorrect data, it is relatively easy to spot incomprehensible data, although finding the source of the problem may be as difficult as it is with incorrect data.

Multiple Values in a Column

Assume that you are working with a personnel database. The dependents table has the following structure:

```
dependents (employee_ID, child_first_name, child_birth_date)
```

The intent, of course, is that there will be one row in the table for each dependent of each employee. However, when you issue a query to retrieve the dependents of employee number 12, you see the following:

```
Employee_ID     child_first_name        child_birth_date
12              Mary, John, Sam         1-15-00
```

Clearly something is wrong with these data. As we discussed earlier in this book, putting multiple values in the same column not only violates the precepts of the relational data model, but makes it impossible to associate data values accurately. Does the birthdate apply to the first, second, or third dependent? Or does it apply to all three? (Triplets, perhaps?) There is no way to know definitively from just the data in the database.

Character columns are particularly vulnerable to multiple data values, especially where names are concerned because they must be left without constraints other than a length. You can't attach constraints that forbid blanks or commas because those characters may be part of a legitimate

constraints other than a length. You can't attach constraints that forbid blanks or commas because those characters may be part of a legitimate name. The only solution to this type of problem is user education: You must teach the people doing data entry that multiple values mean multiple rows.

Orphaned Foreign Keys

So much of the querying that we do of a relational database involves joins between primary and foreign keys. If we delete all foreign key references to a primary key, then the row containing the primary key simply doesn't appear in the result of a join. The same thing occurs if the primary key referenced by a foreign key is missing: The rows with the foreign keys don't appear in the result of a join. The former *may* be a problem, but the latter always is. For example, a customer with no orders in a database may be just fine, but orders that can't be joined to a customer table to provide customer identification will be a major headache.

A relational database must prevent these "orphaned" foreign keys from existing in the database and the solution should be provided when the foreign keys are defined. As you will remember from Chapter 11, the definition of a foreign key can contain an ON DELETE clause. Its purpose is to specify what should happen to the row containing a foreign key when its primary key reference is deleted. The DBMS can forbid the deletion, set foreign key values to null, or delete the foreign key row. Which one you choose, of course, depends on the specific needs of your database environment. Nonetheless, an ON DELETE clause should be set for every foreign key so that orphans never occur.

Recognizing and Handling Inconsistent Data

Inconsistent data are those that are correct and make sense but, when duplicated throughout the database and/or organization, aren't the same. We normalize relations to help eliminate duplicated data, but in large organizations there are often multiple databases that contain information about the same entities. For example, a retail company might have one database with customer information for use in sales and another for use in marketing. If the data are to be consistent, then the name and address of a customer must be stored in exactly the same way in both databases.

Note: The best way to handle inconsistent data is the same, regardless of the type of inconsistent data. The solution is, therefore, presented at the end of this section.

Inconsistent Names and Addresses

When names and addresses duplicate throughout an organization, it's tough to keep the data consistent. By their very nature, name and address columns have few constraints because the variation of the data in those columns is so great. Adding to the problem is the difficulty of detecting when such data are inconsistent: You rarely know until someone complains or there is a need to match data between databases.

Inconsistent Business Rules

Some business rules can be implemented as database constraints. When there are multiple data stores within an organization, those constraints may not be applied consistently. Assume, for example, that the highest salary paid by an organization is $125,000 annually. There is a central personnel database, and smaller personnel databases at each of six satellite offices. The central database contains all personnel data; satellite databases contain data for the site at which they are installed. The satellite databases were designed and installed at the same time. The salary column in the employee table has the correct CHECK clause to limit the salary value. However, IT ran into a problem with that check clause at headquarters because the CEO's salary was $1,500,000. Their solution was simply to modify the check clause to allow for the higher salary. The CEO is an exception to the rule, but once the CHECK clause was modified, *anyone* who was entered into the central database could be given a higher salary without violating a table constraint.

Now assume that the CEO hires a new manager for one of the satellite offices at a salary of $150,000. Someone on the human resources staff enters the new employee's information into the database. Because the constraint on the salary limit has been removed, the data are stored. However, when the update is propagated to the appropriate satellite database, the constraint on the employee table prevents the update. The organization is left with a distributed database in an inconsistent state.

Inconsistent Granularity

Granularity is the level of detail at which data are stored in a database. When the same data are represented in multiple databases, the granularity may differ. As an example, consider the following tables:

```
order_lines (order_numb, item_numb, quantity, cost)

order_lines (order_numb, item_numb, cost)
```

Both tables contain a cost attribute, but the meaning and use of the columns are different. The first relation is used in the sales database and includes details about how many of each item were ordered and the amount actually charged for each item (which may vary from what is in the items table). The second is used by marketing. Its cost attribute is actually the line cost (quantity * cost from the sales database). Two attributes with the same name therefore have different granularities. Any attempt to combine or compare values in the two attributes will be meaningless.

This type of problem needs to be handled at an organizational level rather than at the single database level. See the last part of this section for details.

Unenforced Referential Integrity

In the preceding section, we discussed the problem of orphaned foreign keys. They represent a violation of referential integrity that occurs *after* the data have been stored and can be handled with strict foreign key definitions when the tables are created. However, what happens if foreign key

the data have been stored and can be handled with strict foreign key definitions when the tables are created. However, what happens if foreign key constraints are never added to a table definition? Foreign keys may then be orphaned as soon as they are added to the database because there is nothing to ensure that they reference existing primary keys.

As you might expect, the solution to this problem is straightforward: Ensure that referential integrity constraints are present for all foreign keys. Make this a policy that everyone who has the right to create tables in the database must follow.

Inconsistent Data Formatting

There are many ways to represent the same data, such as telephone numbers and dates. Do you surround an area code with parentheses or do you follow it with a hyphen? Do you store the area code in the same column as the rest of the phone number or is it stored in its own column? Do dates have two or four digit years? Does the month come first, the day, or the year? How are months represented (numbers, codes, full words)? There are so many variations for telephone numbers and dates that unless there is some standard set for formatting throughout an organization's data management, it may be nearly impossible for queries to match values across databases.

Preventing Inconsistent Data on an Organizational Level

There is no easy solution to preventing inconsistent data through an organization. It requires planning at the organizational level and a commitment by all those who are responsible for databases to work together and, above all, to communicate.

Fixing the Problem

A large organization with multiple databases should probably be involved in *data administration*, a process distinct from *database administration*. Data administration keeps track of where data are used throughout an organization and how the data are represented. It provides oversight for data at an organizational level, rather than at the database level. When the time comes to use the data in a database or application program, the developers can consult the *metadata* (data about data) that have been identified through the data administration process and then determine how the data should be represented to ensure consistency.[1]

It is important to keep in mind, however, that even the best data administration can't totally protect against inconsistent names and addresses. Although organizational metadata can specify that the abbreviation for street is always "St." and that the title for a married woman is always stored as "Mrs.", there is no way to ensure that names and addresses are always spelled consistently. Human error will always be a factor. When that occurs, the best strategy may be just to smile sweetly to the complaining customer and fix the problem.

Employees and Data Quality

One recurrent theme throughout this chapter is that many data quality problems are the result of human error. They may be attributable to a single individual or to a group of employees as a whole. But how will you know? The database needs to keep track of who enters data. The easiest way to implement such audit trails is to add a column for an employee ID to each table for which you want to keep data entry data. Assuming that an employee must log in to the system with a unique user name before running any application programs, the application programs can tag rows with the employee ID without any employee intervention.

If you need to keep more detailed modification information, in particular maintaining an audit trail for each modification of a table, then you need a further modification of the database design. First, you give each row in each table a unique numeric identifier:

```
customer (customer_numb, customer_first_name,
    customer_last_name, customer_street, customer_city,
    customer_zip, customer_phone, row_ID)
```

The row ID is an integer that is assigned when a row is created. It is merely a sequence number that has no relationship to the row's position in the table, or to the customer number. The row ID continues to get larger as rows are entered to the table; row IDs for deleted rows are not reused.

Note: Should a table be old enough or large enough to run out of row IDs, row IDs for deleted rows can be reused or row IDs can be reassigned to the entire table in a contiguous sequence (perhaps using a larger storage space, such as a 64-bit rather than 32-bit integer). This is a rather lengthy process, because all foreign keys that reference those row IDs must be updated, as well.

The design must then include a table to hold the audit trail: