2.1 The OLTP Environment

2.1.1 OnLine Transaction Processing

2.1.2 Record Real-world events

2.1.3 Examples

2.1.3.1 Purchasing – revenue cycle

2.1.3.2 Sales – revenue cycle

2.1.3.3 Human Resources

2.1.3.4 Service

2.1.3.5 Inventory

2.1.4 Characteristics

2.1.4.1 Dynamic data

2.1.4.1.1 Brand new data

2.1.4.1.2 Changes to existing data

2.1.4.1.3 Deletes to data

2.1.4.2 High volume of transactions

2.1.4.3 Need for performance

2.1.4.4 Need for accuracy

2.1.4.5 Need for durability

2.1.4.6 Support of multiple systems, multiple users, i.e., concurrency
2.2 Tables and the SELECT statement (controlling column output)

In this chapter, we cover tables and where they come from. Then we begin using SQL to retrieve information from an existing database by using the SELECT statement. All results from a database are tables, and the SELECT statement is used to define the columns that will be in the resulting tables.

2.2.1 Real-world, Logical, Physical

Organizations need to be able to store data about things, people, ideas, and events in the real world. We can plan out how to store this data without knowing the details of how that data is to be physically stored.

For example, a company may be interested in storing data about their customers. They may wish to store customers' names, address, phone numbers, etc. They may also specify that an email address is essential to their business, as is each customer's credit score. In addition, they may need a way to uniquely identify each customer. With these specifications, the company can create a logical data model. See figure 4.1 for a logical model of a customer.

<Figure 4.1 about here>
Logical models are used to brainstorm about the data that is required by the organization, without slowing down to think about the details of how it will be stored. Note that the logical data model is useful, even though it does not specify whether this data will be stored on paper forms, in a spreadsheet, in a database, or pressed into a block of clay.

Logical data models typically capture:

- **Entities**
- **Attributes**
- **Keys**
- **Relationships** (covered in the next chapter)

Entities are things of interest in the real world. A thing of interest might be:

- A person (a customer, an employee, a contact, etc.)
- A physical object (a car, a room, a product, etc.)
- An event (a sale, a registration, a web page view, etc.)
- An idea (an invoice, a contract, a warranty, a bid, etc.)

More specifically, an Entity is a type of a thing. In our example, a Customer is a type of a person. A particular Customer, say, Jill Jackson, would be called an instance of an entity.

Attributes are characteristics of an entity that are of interest to the organization. In the current example, name, email, credit score, and address are all attributes of Customer that are important to the business. Different entities, of course, will have different attributes. The term attribute can be used in two different senses that may cause confusion:

- An attribute of an entity
  - The Customer entity has an email attribute
- A specific attribute value of an entity instance
  - Jill Jackson’s email attribute is “jill@jillscompany.com”

Keys are special attributes whose purpose is to uniquely distinguish an entity instance from all other entity instances. It’s quite common to have multiple entity instances that have identical attributes. For example, it is quite possible to have two different customers name “Jill Jackson”. In fact, as the number of customers we have grows, it’s practically guaranteed that we’ll have two or more customers with identical names.

To address this issue, for each entity, we choose an attribute or combination of attributes that we expect (or even guarantee) to be unique across entity instances. A key for Jill Jackson in a system might look like any of the following:

- Jackson1000
- 98RFX9
As you can see, the key does not need to be meaningful; it merely needs to be unique for Jill Jackson. Many databases simply give each entity instance the next number in a sequence: 1, 2, 3, etc. However, not all keys are created equal: some choices of keys can cause problems in the future, as you’ll see later.

Relationships, which will be covered in the next chapter, indicate how entities are related. For instance, a particular Contract entity instance might be related to a particular Customer entity instance. Relationships are very important to creating high value information from the data in the database.

Logical data models are typically created by experts from the organization, not database experts. For instance, if an insurance company wants to create a database, a logical model should be developed by experts in the insurance business, not by database or technical experts. Think of the logical data model as the first high-level sketch of what the database might look like. Furthermore, it is an easily understood representation that is readable by both database experts and organization experts. However, a logical model is still far from a working database.

Given a logical data model as a starting point, a database architect can design a physical data model. The physical data model adds specifics about how the data will be stored in a relational database. For instance, the Customer entity from our logical data model is likely to be a table in a physical data model.
Figure 4.2 gives a possible physical data model that corresponds to the logical data model from Figure 4.1. Here, Customer is defined to be a table with columns ID, firstName, lastName, etc. Furthermore, the data type and size is specified for each column. Several things to notice about the physical data model in comparison to the logical data model:

- there is no “cookbook” or predefined way to create a physical model from a logical model; it is an art that requires significant insight, experience, and skill
- one attribute might become multiple columns, for example, the name attribute has become firstName, lastName, and middleInitial
- a complex attribute, like address, might be stored in a separate table, with a reference to the appropriate key value

The physical data model can be viewed as a blueprint for a database. A database analyst can take a physical data model, and create a database. Figure 4.3 shows some sample data in tabular form that follows the physical data model for Customer. (Hopefully, the translation from logical to physical model
did not change the intent of the organization experts. A physical model should always be reviewed by an organization expert to make sure it did not change their intention.)

<Figure 4.3 about here>

### 2.2.2 Parts of a table

Tables are the basic representation of data in a relational database. For most people, a table is an intuitive, natural way to organize data. The basic parts of a table are (refer to Figure 4.3):

- Columns
- Rows
- Cells
Columns are the vertical organizing mechanisms for the table, rows are the horizontal organizing mechanisms for the table, and cells are the intersection of the columns and rows. Columns are sometimes called *fields*, or *attributes*. Rows are sometimes called *records*, or entity instances. Cells are sometimes called *column values*, *field values*, or *attribute values*.

In addition to the visible items in the table, there are also several very important pieces of information that are not in the table. The database engine maintains quite a bit of data about tables, called *Metadata*. Examples include:

- Table name
- Column names
- Key column

In Figure 4.3, Customer is the name of the table. The column names are ID, firstName, lastName, etc. Note that column names must be unique in each table, so that they can be referred to by name without confusion. The key column for the table is ID. These are typically not visible “in the table”, but they are essential to using the table.

Several basic “golden” rules are should be applied when using tables to prevent problems with using the tables. These may seem obvious, but are commonly violated, and nearly always create significant pain and cost to the organization.

**Table Golden Rule #1: Field Values in a Column Should Have the Same Meaning for Every Record**

For instance, the firstName column should contain only first names. Never should it contain anything else. The creditScore column should always contain creditScores. Why might anyone put anything else in the creditScore? Here are some examples:

- Someone might put “ZZZZ” in creditScore to indicate that the credit score was unavailable
- Someone might put “Jack (Jay Jay)” in the firstName column to indicate that Jack’s nickname is “Jay Jay”.
- Someone might put “**Jack” in the firstName column to indicate that Jack is a “special” customer
- The creditScore might contain creditScore for customers who are individual people, but corporate bond rating for customers who are companies
- Any host of other oddities

These violations of Table Golden Rule #1 seem inconsequential at first glance. However, they cause the column and its data to need special processing each time the data is used. Each system that uses the column must know all of these oddities, and deal with them, increasing the cost of building and maintaining the system. In short, life is better for everyone when the column’s meaning does not change from row to row.

**Table Golden Rule #2: Field Values in a Row Should Pertain to that Row**
For instance, all field values of row 1 in Figure 4.3 pertain to Jill J. Jackson who has customerID=17. The email address “jill@jillscompany.com” is Jill J. Jackson’s email address and no one others. Why might someone put anything that doesn’t pertain to Jill in Jill’s record? Here are some possible examples:

- Someone might put Jill’s company’s sales department’s email address if hers is unavailable
- Someone might put Jill’s spouse’s creditScore in her row, if hers is unavailable
- Someone might put Jill’s company’s creditScore in her row
- Someone might put “*******” in her firstName field to indicate that that is where they stopped entering data for the day
- Any host of other oddities

Again, these might seem okay at first glance, but can cause significant cost and reduce the potential value of the data. The value of the data is kept high when all fields in a row hold meaning specific to that row.

**Table Golden Rule #3: Each cell should contain a single piece of data**

In our example, the name of each customer was designed to be stored in three cells: firstName, lastName, and middleInitial. However, a different data design could have them all in a single cell. Or a single cell (field) could be used to store both creditScore and annual salary.

This is a less concrete rule than the first two. One might use Rule #3 to say that email address should be stored in three (or more) pieces (and many data architects would agree!) This rule comes from the insight that it is typically easier to combine pieces of data than it is to separate them. For instance, names in the form “Jill J. Jackson” are harder to sort by last name than names in the form “Jill”, “Jackson”, “J”. And it is relatively simple to combine “Jill”, “Jackson”, and “J” into “Jill J. Jackson”.

These three “Golden Rules” are the hard-earned wisdom of professionals through the years. They might be called database “common sense”, although they may not be common sense to non-database professionals. They are relatively simple, but often provide a kind of test for “does it make sense to do this in my database?” To a large degree, they are the basis of relational database theory, data normalization, and database best-practices.

### 2.2.3 SELECT

Structured Query Language (SQL) is the language used to access data in a relational database. The most basic way to access data is through the SELECT statement. Here is a very simple SELECT statement:

```
SELECT 26
```

If you execute this command in SQL Server 2008, the result will look like:
Looks like a 1-column, 1-row table, right? This may seem a bit strange, but it makes an important point that will help later:

The result of a SELECT statement is always a table.

The column does not have a name, because we didn’t give it one. Let’s give the column a name:

```
SELECT 26 AS myNumber
```

Now the column has a name:

Let’s summarize the basic form of the SELECT statement so far:

```
SELECT expression AS columnName
```

The SELECT and AS words are known as reserved words of SQL. This means that they hold a special meaning in SQL, and should not be used for any other purpose (e.g., a column name or table name.)

In the last example above, the expression was 26, and the columnName was `myColumn`. In a SELECT statement, columnName must follow a basic naming convention defined by the database engine. In SQL Server 2008, the columnName must:

- Start with a letter (a-z, A-Z) or underscore(_) or asterisk(*) or at-sign (@).
- Contain only letters, digits, underscore, asterisk, at-sign, dollar sign ($), or number sign (#).
- Not be a reserved word

However, it is possible to break these rules somewhat by delimiting the columnName with quotes. For example:
SELECT 26 AS ‘my Column’
SELECT 26 AS “my Column”
SELECT 26 AS [my Column]

Notice that in all these examples, the space character is used, which is normally not allowable. Single quotes, double quotes, and square brackets can be used as delimiters. In this book, we’ll try to consistently use square brackets, as in the third example.

The other part of the SELECT statement is the expression. Here are some examples of what kinds of things can go into expressions:

SELECT (4 * 7) - 8/3 AS [mathematicExample]
SELECT SQRT(4598.43) AS [functionExample]
SELECT SQRT(9345.2)/6 AS [combinationExample]
SELECT ‘J’ + ‘ill’ AS [charExample]

The examples above are not exhaustive, but give an idea of what can be included in an expression. Most databases have a full set of arithmetic and logical operators, functions and operators for manipulating character data, and a large variety of built-in functions. (And you can create your own functions if what you need is not available.)

The examples above have all generated single-column tables. Here is an example that generates multiple columns:

SELECT 3.1415 AS [PI],
3/2 AS [threeHalfs]

So the more general form of a SELECT statement is:

SELECT expression AS columnName,
expression AS columnName,
...

You can specify multiple columns as long as they are separated by commas, and each columnName is unique. The columnNames need to be unique so that they can be referred to in other expressions without confusion.

The examples above have been actually creating all their own data. Here is an example where data is drawn from a table:

SELECT firstName AS [personName]
FROM Person

Notice the FROM clause that has been added to the SELECT statement. The FROM clause specifies the table from which the firstName column should be drawn. For this to execute without error, the Person table must have a column named firstName.

What are the possible results from this SELECT statement?
A one-column, zero-row table, if there are no records in the table
A one-column, many-row table, if there are records in the table

If the table has 10 rows at the moment the SELECT statement is executed, it will return a 1-column, 10-row table. The column will be named personName, and will contain the firstName field values for the 10 records. If the table has 10,000 rows at the moment the SELECT statement is executed, it will return a 1-column, 10,000-row table.

Will the SELECT statement always return the same table? Only if the data in the table never changes. The SELECT statement will query the database table at that moment, and return the results. In an active OnLine Transaction Processing (OLTP) database, it is unlikely that a SELECT statement from a table will return the exact same table twice.

Another thing to note about the last SELECT statement: the expression was a column name. Not only can a column name be used as an expression, but a column name can be used in an expression:

```
SELECT ageInYears*365 AS [approxDaysOld]
FROM Person
```

Assuming the Person table has an ageInYears column, the above statement uses this column to calculate the approxDaysOld for each row in the table. Will all rows use the same ageInYears? No. Remember that ageInYears is a column, and each row has a value for that column. So each rows’ approxDaysOld will be calculated with its own ageInYears.

In summary, the SELECT statement is mainly used to specify the columns of a table to be retrieved. In addition, it can be used to create or calculate data using values from the table, or independently. In the FROM examples above, all rows from the table are returned. In the next chapter, the WHERE clause will be used to specify exactly which rows should be returned.