2.5 FROM (equi-joins)

2.5.1 Qualifying field names with table names

Consider the Suppliers and Products tables from the Microsoft Northwind reference database. Each of
the tables, Suppliers and Products, specify data about each entity. However, there is a special field:
SupplierID in the Products table. For each record in the Products table, the SupplierID enables you to
look up details about the supplier who supplies this product. In other words, it provides a look-up to the
related Supplier.

By relating data about one entity with data about another entity, new information can be produced that
cannot be produced from the entities separately. As with other data operations such as filtering and
sorting, relating data is a way to produce valuable information from data.

Relational databases explicitly represent relationships using foreign keys. When the primary key from
one table is stored in another table, it is called a foreign key. One way to view this is as a look-up into
another table. In the Suppliers and Products example below, the SupplierID in the Products table is the
foreign key to the primary key field SupplierID in the Suppliers table.

To be able to distinguish fields that have the same name in separate tables, SQL allows for qualifying
a field name with the table name. For instance, the SupplierID in the Products table can be referred to as
Products.SupplierID. Here is an example:

    SELECT Products.SupplierID
    FROM   Products

Which is equivalent to:

    SELECT SupplierID
    FROM   Products
So why is it necessary to qualify field names with table names? Because when multiple tables are used in a SQL statement, field names can be ambiguous, e.g., SupplierID from which table? Here is an example of SQL that is ambiguous, and will generate a syntax error (this kind of a SELECT statement hasn’t been covered yet, but will be shortly):

```sql
SELECT SupplierID -- this is the line with the error
FROM Products, Suppliers
WHERE Products.SupplierID = Suppliers.SupplierID
      -- this will generate a syntax error
```

This generates a syntax error because there is a SupplierID field in both the Products and the Suppliers table. The database engine can’t know which one is intended from the SQL statement. To get rid of the ambiguity, we qualify each field name with the table name:

```sql
SELECT Products.SupplierID, Suppliers.SupplierID
FROM Products, Suppliers
WHERE Products.SupplierID = Suppliers.SupplierID
```

In this example, there is no question which field is being specified.

### 2.5.2 Relationships

Note the line between the Products and Suppliers tables: on one end is a yellow key symbol, on the other end is the infinity symbol. The line specifies a relationship between the Products and Suppliers tables. This can be read:

> For each Supplier, there can be many related Products. For each Product, there is at most a single related Supplier.

More specifically, it also defines the fields involved in the relationship. Another way to state the relationship that includes the field involvement:

> Products.SupplierID is the foreign key to the Suppliers table.

Of course, the ways that entities relate are specified by the organization, not the database. The database should merely reflect the way the organization works. In the Suppliers and Products model, the organization has specified that there is a single supplier for each product. Other organizations might specify that there can be more than one supplier for each product. *The database should reflect the organization’s needs.*

Notice that:

- The foreign key is on the *many-side* (infinity side) of the relationship
- The primary key is on the *one-side* (yellow key) of the relationship
Another way to think of the one-side and the many-side is to think about the field values in the related fields. For example, suppose that we have a certain supplier, with SupplierID=17. Now answer these questions:

- How many times could the SupplierID=17 appear in the Suppliers table? (only once)
- How many times could the SupplierID=17 appear in the Products table? (zero, 1, or many)

SupplierID=17 can only appear once in the Suppliers table, because SupplierID is the primary key of the Suppliers table, and is therefore unique. But SupplierID is not the primary key in the Products table, so it isn’t unique, and can therefore appear many times. In plain English, a single Supplier can supply many products.

2.5.3 Referential Integrity

Do you need a relational database to relate records in one table with records in another table? Not really. The Products.SupplierID field is really just a field that contains values for the purpose of looking up suppliers in the Suppliers table. No other special capability is necessary for this. However, what would happen if a record in the Products table has Products.SupplierID=38, and there is no record with Suppliers.SupplierID=38 in the Suppliers table? The Products.SupplierID is still a lookup, it just doesn’t have a sensible value – it refers to a supplier that doesn’t exist in the Supplier table.

Relational databases provide a special mechanism for making sure that lookup values make sense: referential integrity. Referential integrity enforces that values in the foreign key must match a value in the primary key table. It does this by checking that lookups make sense when certain operations occur:

- When a record is inserted in Products, the SupplierID value is checked to make sure it exists in the Suppliers table
- When a record in Products has its SupplierID value changed, the new value is checked to make sure it exists in the Suppliers table
- When a record in Suppliers is deleted, all the Products records are checked to make sure they did not reference the supplierID value that is being deleted
- When a record in Suppliers has its SupplierID value changed, all the Products records are checked to make sure they did not reference the supplierID value

All of these checks are performed in order to maintain the integrity and quality of the data. If there are any violations, then the operation (insert, update, delete) is blocked. This can be frustrating to users sometimes, because they are blocked from performing their work. However, referential integrity is really their friend, because it helps them enter accurate information.

2.5.4 Joining with the WHERE clause

In most cases, the databases you will work with are already established. The database tables and relationships were designed by business analysts and database analysts. As the organization goes about its daily business, data is inserted into the tables, and the database maintains consistency in the
relationships by enforcing referential integrity. This enables decision makers to relate data between tables to create information, using a SELECT SQL statement.

As an example, consider some sample data for the Products and Suppliers tables shown earlier.

<table>
<thead>
<tr>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductID</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suppliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SupplierID</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>103</td>
</tr>
</tbody>
</table>

In the Products table are five records with values for SupplierID that refer to records in the Suppliers table. Note that there are no Products that are supplied by SupplierID=102, Atlantic Metal. Based on the first record in the Products table, we know that ProductID=1 is a ball point pen, and is supplied by SupplierID=103, which is Smith Office Supply.

There are two ways to accomplish relating data between two tables. The first uses the WHERE clause, and looks like this:

```sql
SELECT Products.ProductID,
       Products.ProductName,
       Products.SupplierID,
       Suppliers.SupplierID,
       Suppliers.CompanyName
FROM Products,
     Suppliers
WHERE Products.SupplierID = Suppliers.SupplierID
```

To understand exactly what this means, consider what would happen in the above SELECT statement if the WHERE clause were not included:

```sql
SELECT Products.ProductID,
       Products.ProductName,
       Products.SupplierID,
```

© 2010, Douglas M. Kline  Do Not Copy, Personal Use Only
Suppliers.SupplierID,
Suppliers.CompanyName
FROM Products,
Suppliers
-- returns all possible combinations of Products
-- and Suppliers
-- not normally what you want

Without the WHERE clause, all possible combinations of the records in Products and Suppliers will be returned:

<table>
<thead>
<tr>
<th>Products. ProductID</th>
<th>Products. ProductName</th>
<th>Products. SupplierID</th>
<th>Suppliers. SupplierID</th>
<th>Suppliers. CompanyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ball Point Pen</td>
<td>103</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>1</td>
<td>Ball Point Pen</td>
<td>103</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>1</td>
<td>Ball Point Pen</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>2</td>
<td>White Copy Paper</td>
<td>103</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>2</td>
<td>White Copy Paper</td>
<td>103</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>2</td>
<td>White Copy Paper</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>3</td>
<td>Screwdriver</td>
<td>101</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>3</td>
<td>Screwdriver</td>
<td>101</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>3</td>
<td>Screwdriver</td>
<td>101</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>4</td>
<td>Hammer</td>
<td>101</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>4</td>
<td>Hammer</td>
<td>101</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>4</td>
<td>Hammer</td>
<td>101</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>5</td>
<td>Printer Toner</td>
<td>103</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>5</td>
<td>Printer Toner</td>
<td>103</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>5</td>
<td>Printer Toner</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
</tbody>
</table>

The database engine creates all possible combinations of Products records and Suppliers records, with no regard to any values in those records. In this case, there are five records in Products, and 3 records in Suppliers, so 3 X 5 = 15 records are returned. This is called a cross join. In large tables, the number of records returned could be quite large, and create a strain on the database server that has to create the result.

In some tasks that database administrators perform, the cross join can be helpful. But for decision makers, showing unrelated records is not helpful. The decision maker typically only wants to see the records that are related. These are in the cross join results shown above, but are mixed in with results that show unrelated records. The results that we are interested in are where the Products.SupplierID is equal to the Suppliers.SupplierID. The rows of related records are emphasized below:
To limit the rows returned to just the ones we want, we can use the WHERE clause, as in our first example:

```
SELECT Products.ProductID,
       Products.ProductName,
       Products.SupplierID,
       Suppliers.SupplierID,
       Suppliers.CompanyName
FROM   Products,
       Suppliers
WHERE  Products.SupplierID = Suppliers.SupplierID
```

Which will return:

<table>
<thead>
<tr>
<th>Products. ProductID</th>
<th>Products. ProductName</th>
<th>Products. SupplierID</th>
<th>Suppliers. SupplierID</th>
<th>Suppliers. CompanyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ball Point Pen</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>2</td>
<td>White Copy Paper</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>3</td>
<td>Screwdriver</td>
<td>101</td>
<td>101</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>4</td>
<td>Hammer</td>
<td>101</td>
<td>102</td>
<td>Atlantic Metal</td>
</tr>
<tr>
<td>5</td>
<td>Printer Toner</td>
<td>103</td>
<td>103</td>
<td>Smith Office Supply</td>
</tr>
</tbody>
</table>
Of course, decision makers would be more interested in product names, and company names. A more typical SELECT statement for a decision maker might be:

```sql
SELECT Products.ProductName,
       Suppliers.CompanyName
FROM   Products,
       Suppliers
WHERE  Products.SupplierID = Suppliers.SupplierID
```

<table>
<thead>
<tr>
<th>Products. ProductName</th>
<th>Suppliers. CompanyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball Point Pen</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>White Copy Paper</td>
<td>Smith Office Supply</td>
</tr>
<tr>
<td>Screwdriver</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>Hammer</td>
<td>Jones Tools</td>
</tr>
<tr>
<td>Printer Toner</td>
<td>Smith Office Supply</td>
</tr>
</tbody>
</table>

### 2.5.5 Joining with the JOIN operator

The WHERE clause syntax was the first standard syntax for relating records in a SELECT statement. However, it doesn’t represent the fact that this is not a simple criteria that we’ve seen in the WHERE clause before. This criteria is special because it relates foreign keys to primary keys. A newer syntax separates the joins from the simple WHERE criteria:

```sql
SELECT Products.ProductName,
       Suppliers.CompanyName
FROM   Products
       JOIN Suppliers ON Products.SupplierID = Suppliers.SupplierID
```

In this syntax, the foreign key – primary key join condition is stated in the FROM clause using the JOIN...ON operator. This will always return exactly the same records that the WHERE clause syntax will return. Because it separates the join condition from simple WHERE constraints, it is sometimes easier to read. This syntax also allows for qualifying the JOIN...ON operator to specify which type of JOIN (some very important JOINs will be covered later). For example, here is the syntax for the cross join we showed earlier, and how it would be done with the JOIN...ON operator:

```sql
SELECT Products.ProductName,
       Suppliers.CompanyName
FROM   Products,
       Suppliers
--does the cross join, but was it a mistake?
```
SELECT Products.ProductName, Suppliers.CompanyName
FROM Products
CROSS JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID
-- no mistake, we really meant to do a cross join

The main advantage here is that by typing in CROSS JOIN, we indicate that we really meant to do a cross join. The first cross join looks as if there might have been a mistake – it doesn’t explicitly state that a cross join was intended. In short, the JOIN syntax better documents what the SQL author intended.

2.5.6 Many-to-Many JOINs

Consider the following data model:

The ClassMember table in this data model plays a special role. It **associates** Students with ClassSections. More specifically, it associates records from the Student table with records from the ClassSection table. This is done by having a foreign keys from both the Student and ClassSection table.

How is this different than the association between Major and Student? In the data model above, a Student can have only one Major. A Major can have more than one student in it. The Major-to-Student relationship is an example of a **one-to-many** relationship.

Now consider the Student-to-ClassSection relationship. A Student can be in more than one ClassSection. A ClassSection can have more than one Student in it. The Student-to-ClassSection relationship is an example of a **many-to-many** relationship. So one of the purposes of the ClassMember table is to associate Students with ClassSections in a many-to-many way. Tables like ClassMember that represent many-to-many relationships are called **associative** entities.

Note that there’s nothing new here. Many-to-many relationships are nothing more than two one-to-many relationships. Everything you know about one-to-many relationships applies: foreign keys, referential integrity, JOINing, etc.

Let’s look at some data to clarify:
The Student table should be self-explanatory. The first record in the ClassSection table indicates that there is a class in the Fall term of 2012. The subject of that class is “Management”, indicated by the MGT subject code. The courseNumber of the class is 105, and there appears to be only one class section, indicated by the section Number 001.

Here is what the records in the ClassMember table tell us, by ID:

- 1: John Smith (studentID=1) is in MGT 105 section 001(classSectionID=1)
- 2: John Smith is also in MGT 210 section 001
- 3: Charles Brown is in MGT 105 section 001
- 4: Jane Doe is in MGT 105 section 001
- 5: Jane Doe is in MGT 210 section 002
So the ClassMember table enables us to understand how Students relate to ClassSections. Better yet, the information in the bullet list above can be retrieved by using a SELECT statement and JOINing the Student, ClassMember, and ClassSection tables:

```
SELECT     Student.firstName,
            Student.lastName,
            ClassSection.subjectCode,
            ClassSection.courseNumber,
            ClassSection.sectionNumber
FROM       Student
           JOIN ClassMember  ON Student.ID = ClassMember.studentID
           JOIN ClassSection ON ClassSection.ID = ClassMember.sectionID
WHERE      ClassSection.year = 2012
           AND     ClassSection.term = 'Fall'
ORDER BY   ClassMember.ID
```

This SELECT statement presents the same information as in the bullet list above. Technically, there is no new concept here; the JOIN statement is the same as presented earlier. The only new concept is that it is possible to JOIN multiple tables in a single SELECT statement. It is also possible to do this with the WHERE clause form of a JOIN:

```
SELECT     Student.firstName,
            Student.lastName,
            ClassSection.subjectCode,
            ClassSection.courseNumber,
            ClassSection.sectionNumber
FROM       Student,
            ClassMember,
            ClassSection
WHERE      Student.ID = ClassMember.studentID
           AND     ClassSection.ID = ClassMember.sectionID
           AND     ClassSection.year = 2012
           AND     ClassSection.term = 'Fall'
ORDER BY   ClassMember.ID
```

It is also possible to JOIN more than three tables. In general, if you are combining three tables, you’ll need two JOINs. If you are combining eight tables, you’ll need seven JOINs.

Be careful! You don’t want to forget a JOIN! If you forget a JOIN, you’ll get all possible combinations of records between the missing JOIN. This is potentially a lot of records, which can disable the database server while it is working on your SELECT statement.