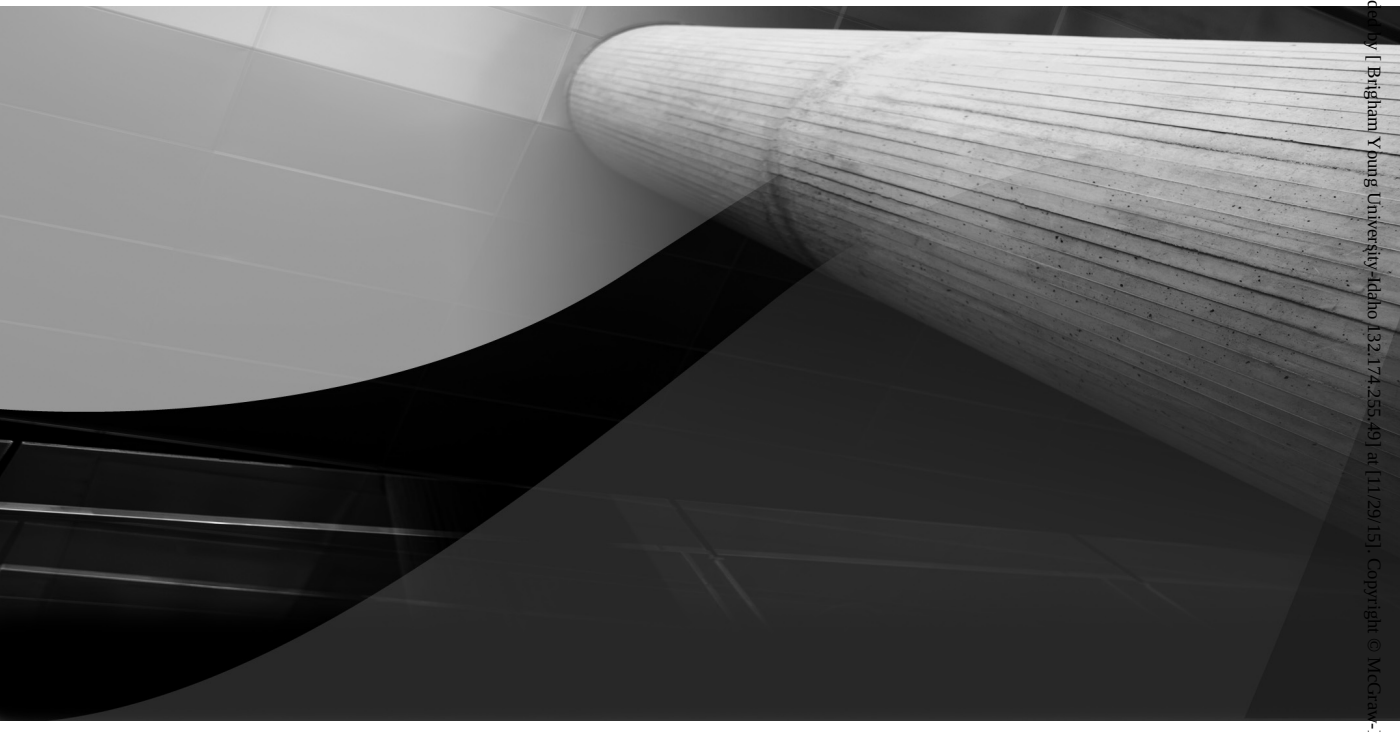


CHAPTER 5

Constraints

onstraints are critical components in databases. They *restrict* (constrain) what you can add, modify, or delete from tables with `INSERT`, `UPDATE`, and `DELETE` statements. Constraints let you restrict what can be placed in columns, rows, tables, or relationships between tables. That's a tremendously broad statement that requires some qualification. So what are these restrictions and why are they important?

As with the prior chapter, this chapter develops what and why before delving into the SQL implementation of database constraints that work with columns, tables, and relationships, which are covered in Chapters 6 and 7. Those chapters show you, respectively, how to implement constraints as you create new structures or modify them through maintenance activities. Chapters 6 and 15 cover constraints that restrict row behaviors, which include the `CHECK` constraint in an Oracle database and triggers in both Oracle and MySQL databases.

This chapter covers the following database constraints; database triggers are also included, because they hold the keys to advanced constraints:

- `NOT NULL`
- `UNIQUE`
- Primary key
- Foreign key
- `CHECK`
- Trigger

A preliminary understanding of constraint capabilities should help you focus on their respective roles as you read this chapter. Two types of constraints are used: column and table constraints. A column constraint applies to a single column. You define it in-line by adding it to the same line in which you define the column. In-line constraints don't require an explicit reference to the column because they apply to the column that shares the line. You can also define a column constraint after defining of all column values in both an Oracle and MySQL database. Constraints that follow the definition of columns are out-of-line constraints, because they are not placed on the same line as their column definition.

You create tables with the `CREATE` statement, covered in Chapter 6. The generalized definition of a `CREATE TABLE` statement is as follows:

```
CREATE TABLE table_name
( column_name data_type [inline_constraint] [, ...]
[, out_of_line_constraint [, ...]);
```

Some constraints involve more than a single column. Multiple-column constraints are *table constraints*. Table constraints are always defined after the list of columns as you create a table, because they depend on the column definitions. Alternatively, table constraints can be added to a table definition with an `ALTER TABLE` statement. Chapter 7 covers the syntax for maintenance activities such as the `ALTER TABLE` statement.

Figure 5-1 provides a matrix that compares constraints against the behaviors they can restrict: columns, rows, tables, and external relationships between tables.

Column-level constraints let you restrict whether a column can be empty or must contain a value. They also let you restrict the values that can be inserted into a column, such as only a *Yes* or *No* string, and restrict the values to a list of values found in another table (that's the role of a foreign key, as you'll discover later in this chapter).

Constraint	Column	Row	Table	External
Not Null	✓	✗	✗	✗
Check	✓	✓	✗	✗
Unique	✗	✗	✓	✗
Primary Key	✓	✗	✓	✗
Foreign Key	✓	✗	✗	✓
Index	✗	✗	✓	✗
Trigger	✓	✓	✓	✓

FIGURE 5-1. *Constraint matrix*

Row-level constraints let you restrict the behavior of one or a group of column values in the same row. For example, you could constrain one column's value based on another column's value.

Table-level constraints let you restrict the behavior between rows in a table. A unique constraint on one or a group of columns prevents more than one row from having the same value for that column or group of columns.

External constraints are trickier, because they involve relationships between tables. They limit the value list of a column or a group of columns (foreign key) to those values already found in another column or group of columns (primary key) in another table or a copy of the same table. This type of constraint is known as a *referential integrity constraint* because it ensures that a reference in one table can be found in another.

The following sections qualify not null, unique, primary key, foreign key, and database trigger constraints. The sections describe differences between constraints in Oracle and MySQL databases.

NOT NULL Constraints

A `NOT NULL` constraint applies to a single column only, as shown in Figure 5-1. It restricts a column by making it mandatory, which means you can't insert a row without a value in the column. Likewise, you can't update a mandatory column's value from something to a null value. Optional columns are nullable or null-allowed. This means you can enter something or nothing, where nothing is a null value. Null values also differ from empty strings in MySQL.

In MySQL, the `CREATE TABLE` statement lets you assign a `NOT NULL` column constraint as in-line or out-of-line constraints. Oracle requires that you define `NOT NULL` constraints in-line. There's no option to add a `NOT NULL` constraint as a table constraint in an Oracle `CREATE TABLE` statement. You can, however, use the `ALTER TABLE` statement to add a `NOT NULL` constraint to an existing table's columns in both Oracle and MySQL databases.

`NOT NULL` constraints impose a minimum cardinality of `1` on a column, which typically makes the column's cardinality [1..1] (one-to-one). This is a Unified Modeling Language (UML)

notation for cardinality. The UML notation assigns the minimum cardinality constraint to the number on the left and the maximum cardinality constraint to the number on the right. The two dots in the middle indicate a range.

Maximum cardinality is always considered one, because each column has one data type and one value in a relational model. The rule applies to all scalar data types.

The maximum cardinality rule changes in an object relational database management system (ORDBMS), such as an Oracle database. That's because it supports collection data types. In an ORDBMS, the maximum cardinality can be one to many, depending on what you measure. It is one when you measure whether a column contains a collection data type or not; and it's many when you measure the number of elements in a collection data type. Another twist is some arbitrary number between one and many, which happens with a `VARRAY` data type in an Oracle database. The `VARRAY` has a fixed maximum size like ordinary arrays in programming languages.

Some implementation differences exist between `NOT NULL` constraints in the Oracle and MySQL databases. The differences qualify where you can define them and how and when you can name them.

Cardinality

Cardinality comes from set mathematics and simply means the number of elements in a set. For example, in an arbitrary set of five finite values, a cardinality of [1..5] qualifies the minimum of 1 and the maximum of 5. This set expresses a range of five values.

In databases, cardinality applies to the following:

- The number of values in an unconstrained column within a row has a default cardinality of [0..1] (zero-to-one) for nullable columns. (The minimum cardinality of zero applies only to nullable columns.)
- The number of values in a `NOT NULL` constrained column within a row has a cardinality of [1..1] (one-to-one).

When there's no upward limit on the number of values in a column, it holds a *collection*. Collections typically contain [1..*] (one-to-many) elements and their cardinality is [0..*] (zero-to-many) in an unconstrained column.

Developers often describe the frequency of repeating values in a table as having low or high cardinality. *High cardinality* means the frequency of repeating values is closer to unique, where unique is the highest cardinality. *Low cardinality* means values repeat many times in a table, such as a gender column where the distribution is often close to half and half. A column that always contains the same value, which shouldn't occur, is in the lowest cardinality possible.

Cardinality also applies to binary relationships between tables. Two principal physical implementations of binary relationships exist: one-to-one and one-to-many. The one-to-many relationship is the most common pattern. In this pattern, the table on the one side of the relationship holds a primary key and the table on the many side holds a foreign key. In a one-to-one relationship, you choose which table holds the primary and foreign key.

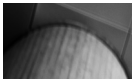
Oracle NOT NULL Constraints

Oracle lets you create NOT NULL columns when you create tables and modify a column in an existing table to make it a NOT NULL or null-allowed column. You perform the former with the CREATE TABLE statement and the latter with the ALTER TABLE statement. Chapters 6 and 7 show you how to use the CREATE TABLE and ALTER TABLE statements.

All rows must be empty or contain data in the target column before you can add a NOT NULL constraint. You can remove a NOT NULL constraint from a column by using an ALTER TABLE statement.

You can mimic the behavior of a NOT NULL constraint by adding a CHECK constraint after the table is created. NOT NULL and CHECK constraints are stored exactly alike in the data catalog. Unfortunately, a NOT NULL restriction on a CHECK constraint isn't displayed when you describe the table. Using a CHECK constraint to mimic a NOT NULL constraint is not a good idea, because it can be misleading to other developers and disguise business rules.

You can also give meaningful names to NOT NULL constraints in an Oracle database when you create tables. These meaningful names help diagnose runtime violations of the constraint more easily than working with system-generated names.



NOTE
Chapter 6 shows syntax examples for creating columns with or without a NOT NULL constraint. Chapter 7 shows how to add and remove a NOT NULL constraint to or from an existing column.

Finding the name of a NOT NULL constraint is more difficult if you didn't assign a constraint name. You can also find the columns of a NOT NULL or CHECK constraint in the ALL_, DBA_, or USER_CONSTRAINTS and USER_CONS_COLUMNS view. You can use the following query to discover information about constraints:

```
COLUMN owner           FORMAT A10
COLUMN constraint_name FORMAT A20
COLUMN table_name      FORMAT A20
COLUMN position        FORMAT 9
COLUMN column_name     FORMAT A20
SELECT   ucc.owner
,        ucc.constraint_name
,        ucc.table_name
,        ucc.column_name
,        ucc.position
FROM     user_constraints uc INNER JOIN user_cons_columns ucc
ON       ucc.owner = uc.owner
AND      ucc.constraint_name = uc.constraint_name
WHERE    uc.constraint_type = 'C';
```

The same query works to return CHECK constraints, because NOT NULL constraints are variations on CHECK constraints in the data catalog.

MySQL NOT NULL Constraints

MySQL lets you create a table with `NOT NULL` columns. As with the Oracle database, MySQL lets you add and remove `NOT NULL` constraints to or from a column in an existing table. You can add a `NOT NULL` constraint when data exists in the table, provided no null values exist in the target column. There's no need to discuss why you should avoid a `CHECK` constraint, because they don't exist in MySQL.

UNIQUE Constraints

A `UNIQUE` constraint is a table-level constraint, because it makes the value in a column or set of columns unique within the table. Table-level constraints apply to relationships between columns, sets of columns, or all columns in one row against the same columns in other rows of the same table. `UNIQUE` constraints are out-of-line constraints set after the list of columns in a `CREATE` statement. Alternatively, you can add them through an `ALTER` statement after creating a table.

Every well-designed table should have a minimum of two unique keys: natural (covered in Chapter 4) and surrogate keys. The *natural key* is a column or set of columns that describes the subject of the table and makes each row unique. You can search a table for a specific record by using the natural key in a `WHERE` clause (see Chapter 11 for more on queries), which makes them internal identifiers within the set of data in a table. Natural keys are rarely a single column.

All *surrogate keys* are uniquely indexed as a single column. Surrogate keys are ID columns. They're generally produced from automatic numbering structures known as *sequences*. Chapter 6 describes sequences in detail. Surrogate keys don't describe anything about the data in the table. They do, however, provide a unique identifier that can be shared with other related tables. Then those related tables can link their data back to the table where the surrogate keys are unique.

The natural and surrogate keys are potential candidates to become the primary key of a data table. As such, they're also *candidate* keys. The primary key uniquely identifies rows in the table and must contain a unique value, as opposed to a null value. As a rule, you should choose the surrogate key as the primary key, because all joins will use the single column. This makes writing joins in SQL statements easier and less expensive to maintain over time, because surrogate keys shouldn't change or be reused. By itself, a surrogate key, a sequence-generated value, doesn't provide optimal search performance when you have lots of data. That's accomplished by a unique index made up of the surrogate key and the natural key. That type of index helps optimize databases to find and retrieve rows faster.

A `UNIQUE` constraint can apply to either a single column or a set of columns. You can create a `UNIQUE` constraint in Oracle or MySQL when you create or alter a table. The `UNIQUE` constraint automatically creates an index to manage the constraint. After all, it is a table constraint, and when you attempt to add a row that duplicates a unique column or set of columns, there must be a reference against which it can make a comparison to prevent it. Those reference points are *indexes*, and they're organized by a *B-Tree*, an inverted tree structure that expedites finding a matching piece of data. It brackets elements in groups and then subgroups until it arrives at the basic elements of data, which are the column or columns of data qualified as unique.

The next two sections qualify where you can find `UNIQUE` constraints. They also qualify some rules that govern how you can interact with them during creation and removal with the `CREATE TALBE` and `DROP INDEX` statements, respectively.

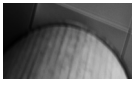
Oracle UNIQUE Constraints

As mentioned, you can create a table with a UNIQUE constraint or alter an existing table to add a UNIQUE constraint. Creating a UNIQUE constraint implicitly adds a unique index. The UNIQUE constraint is visible in the ALL_, DBA_, or USER_CONSTRAINTS administrative view of the database catalog. You can also find the columns of the UNIQUE constraint in a join of the ALL_, DBA_, or USER_CONSTRAINTS and USER_CONS_COLUMNS views. Likewise, you can find another entry for the UNIQUE constraint in the ALL_, DBA_, or USER_INDEXES and USER_IND_COLUMNS views.

The following query shows you how to check the Oracle database catalog for UNIQUE constraints:

```
COLUMN owner           FORMAT A10
COLUMN constraint_name FORMAT A20
COLUMN table_name      FORMAT A20
COLUMN position        FORMAT 9
COLUMN column_name     FORMAT A20
SQL> SELECT   ucc.owner
  2  ,        ucc.constraint_name
  3  ,        ucc.table_name
  4  ,        ucc.position
  5  ,        ucc.column_name
  6  FROM     user_constraints uc INNER JOIN user_cons_columns ucc
  7  ON       ucc.owner = uc.owner
  8  AND      ucc.constraint_name = uc.constraint_name
  9  WHERE    uc.constraint_type = 'U';
```

The query returns a list of all UNIQUE constraints from the data catalog, and each UNIQUE constraint creates a UNIQUE INDEX. You cannot drop this implicitly created index because the UNIQUE constraint is dependent on it. An attempt to drop an implicitly created unique index results in an ORA-02429 exception. This exception's error message text aptly says that you cannot drop an index used for enforcement of a unique/primary key. However, you can alter the table and drop the UNIQUE constraint. The command also implicitly drops the supporting index.



NOTE
Chapters 6 and 7 contain sample SQL statements that let you create UNIQUE constraints in and drop UNIQUE constraints from tables. Chapter 7 also demonstrates how you can modify constraints within the ALTER TABLE statement.

MySQL UNIQUE Constraints

MySQL differs from Oracle in that no separate data catalog tables separate a UNIQUE constraint from a unique index. Like Oracle, the MySQL database lets you create a table with or modify a table to include a UNIQUE constraint. It is made visible only with the following command:

```
SHOW INDEXES { IN | FROM } table_name;
```

The IN or FROM keywords are synonymous. This provides a list of all unique and non-unique indexes that operate against a table.

The information_schema allows you visibility into the data catalog. You can find the UNIQUE constraints with the following query:

```
SELECT   tc.constraint_name
,        tc.constraint_type
,        kc.table_name
,        kc.ordinal_position
,        kc.column_name
FROM     table_constraints tc JOIN key_column_usage kc
ON       kc.constraint_schema = tc.constraint_schema
AND      kc.constraint_name = tc.constraint_name
AND      kc.table_schema = tc.table_schema
AND      kc.table_name = tc.table_name
WHERE    tc.constraint_type = 'UNIQUE'
ORDER BY 1,2,3,4;
```

The query returns a list of all UNIQUE keys and indexes with the participating columns in order. This gives you a snapshot of all UNIQUE constraints in the MySQL server.

Although there's no way to drop any index directly, both indexes and UNIQUE constraints are removed by altering the table and dropping the UNIQUE constraint or index from the table. UNIQUE constraints are dropped by using the KEY keyword, and indexes are dropped by using the INDEX keyword. Chapter 6 and 7 also contain examples to create and drop UNIQUE constraints in MySQL.

Unique Indexes

It is possible to create a unique index that stands apart from a UNIQUE constraint in both the Oracle and MySQL databases. These indexes maintain the same restriction as a UNIQUE table-level constraint because they create indexes to maintain uniqueness among rows in the table. The next two sections qualify the implementation specific details of unique indexes in Oracle and MySQL databases.

Oracle Unique Indexes

You can create an index as a standalone object in an Oracle database. Indexes behave differently than constraints. For example, there is no UNIQUE constraint visible in the USER_CONSTRAINTS administrative view of the database catalog or in the super user views of ALL_ and DBA_CONSTRAINTS. However, you can find entries for unique indexes in the ALL_, DBA_, or USER_INDEXES and USER_IND_COLUMNS views.

This is similar to the query that finds UNIQUE constraints but it uses different views. It finds all unique and non-unique indexes:

```
COLUMN table_owner     FORMAT A10
COLUMN index_name      FORMAT A20
COLUMN table_name      FORMAT A20
COLUMN column_position FORMAT 9
COLUMN column_name     FORMAT A20
SQL> SELECT   ui.table_owner
  2  ,        uic.index_name
  3  ,        ui.uniqueness
  4  ,        uic.table_name
```

```
 5  ,         uic.column_position
 6  ,         uic.column_name
 7  FROM      user_indexes ui JOIN user_ind_columns uic
 8  ON        uic.index_name = ui.index_name
 9  AND       uic.table_name = ui.table_name
10  WHERE     ui.uniqueness = 'UNIQUE';
```

You would find the non-unique indexes with the following change of line 10:

```
10  WHERE     ui.uniqueness = 'NONUNIQUE';
```

You also have the right to drop (or remove) indexes without modifying the table that the indexes organize. This is possible because no `UNIQUE` constraint is dependent on the unique index.

MySQL Unique Indexes

A MySQL unique index is indistinguishable from a `UNIQUE` constraint. In fact, you use virtually the same syntax to create or drop the index. Chapter 6 shows you how to create indexes and Chapter 7 shows you how to drop unique indexes. Feel free to flip there if you want to see the syntax now, but the differences are in the words `KEY` and `INDEX`. A `CREATE` statement that adds a `UNIQUE` table constraint excludes the word `INDEX`, while a constraint of a unique index uses the word `INDEX`.

You can use the query from the "MySQL UNIQUE Constraints" section because keys and indexes are stored in the same tables.

Primary Key Constraints

As mentioned, primary keys uniquely identify every row in a table. Primary keys are also the published identifier of tables. As such, primary keys are the point of contact between data in one table and that in other tables. Primary keys also contain the values that foreign key columns copy and hold. When using referential integrity, the primary and foreign keys shared values let you link data from different tables together through join operations.

Primary keys can be column or table constraints. They're column constraints when they apply to a single column, such as a surrogate key. They're generally table constraints when they apply to a natural key because natural keys usually contain more than one column. Natural keys often contain multiple columns, because that is generally how you qualify uniqueness in a record set.

A single column primary key exhibits two behaviors; these are *not null* and *unique*. A multiple column primary key can have a set of behaviors different from those of a single column primary key. Although the collection of columns must be not null and unique in the set, it is possible that one or more, but not all, columns can contain a null value. This rule is *not* consistently enforced across relational databases in the industry.

The next two sections qualify implementation rules for primary keys in Oracle and MySQL databases. They also show you how to query the status of constraints in the data catalog.

Oracle Primary Key Constraints

Oracle implements all primary keys as `NOT NULL` and `UNIQUE`. This means all columns in a single or multiple column primary key are *mandatory* columns. Any attempt to insert a null value in a column of a primary key generates an ORA-01400 error. The error message tells you that you cannot insert `NULL` into the primary key.

You can assign a meaningful name to primary key constraints, but Oracle assigns a system-generated name when you don't. It is much more difficult to trace back errors on primary key constraints unless you give them meaningful names. You can look up the definition of primary keys in the ALL_, DBA_, or USER_CONSTRAINTS and USER_CONS_COLUMNS administrative views. Primary keys always have a 'P' in the constraint_type column.

Here's the syntax for this query:

```
COLUMN owner           FORMAT A10 HEADING "Owner"
COLUMN table_name      FORMAT A20 HEADING "Table Name"
COLUMN constraint_name FORMAT A20 HEADING "Constraint Name"
COLUMN column_name     FORMAT A20 HEADING "Column Name"
COLUMN constraint_type FORMAT A1  HEADING "Primary|Key"

SQL> SELECT   ucc.owner
  2  ,        ucc.constraint_name
  3  ,        ucc.table_name
  4  ,        ucc.position
  5  ,        ucc.column_name
  6  FROM     user_constraints uc INNER JOIN user_cons_columns ucc
  7  ON       ucc.owner = uc.owner
  8  AND      ucc.constraint_name = uc.constraint_name
  9  WHERE    uc.constraint_type = 'P';
```

MySQL Primary Key Constraints

MySQL implements primary keys differently based on the type of engine that creates the table. Chapter 1 qualifies that MySQL 5.5 uses the InnoDB engine by default, because it is transactional. The InnoDB engine implements single and multiple column primary keys as UNIQUE and NOT NULL constrained. However, there's a twist to that implementation. An empty or zero length string isn't considered a null in MySQL. This makes it possible to insert an empty string in a character column when the column is part of or the only column in a primary key.

```
SELECT   tc.constraint_name
,        tc.constraint_type
,        kc.table_name
,        kc.ordinal_position
,        kc.column_name
FROM     table_constraints tc JOIN key_column_usage kc
ON       kc.constraint_schema = tc.constraint_schema
AND      kc.constraint_name = tc.constraint_name
AND      kc.table_schema = tc.table_schema
AND      kc.table_name = tc.table_name
WHERE    tc.constraint_type = 'PRIMARY KEY'
ORDER BY 1,2,3,4;
```

You can't assign a meaningful name to a primary key constraint in MySQL because it's always PRIMARY by default. The primary key is maintained just like other constraints. You also see a 'PRI' (abbreviation) in the KEY column when you describe a table.

Foreign Key Constraints

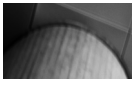
A foreign key constraint is both a column-level and external constraint. The column-level constraint restricts the list of values to those found in a primary key column or set of columns. The primary key column(s) generally exist in another table, which is why an external constraint exists. A self-referencing relationship occurs when the foreign key points to a primary key in the same table. In a self-referencing relationship, the primary and foreign keys are different columns or different sets of columns.

A foreign key constraint basically instructs the database to allow only the insertion or update of a column's value to a value found in a referenced primary key. Foreign keys always contain the same number of columns as the primary key, and the data types of all columns must match. The column and data type matching criteria is the minimum matching criteria. The values in the foreign key column(s) must match the values in the primary key column(s). More or less, foreign keys impose a boundary range of values on foreign key column(s).

The matching values in the foreign and primary key columns allow you to join rows found in one table to those found in another table. Joins between primary and foreign keys are made on the basis of equality of values between two columns or two sets of columns. These are called *equijoins*. (Check Chapter 11 for join possibilities and syntax.)

Foreign key constraints make the database responsible for enforcing cross-referencing rules. This cross-referencing is *referential integrity*, which means that a constraint reference guarantees a foreign key value must be found in the list of valid primary key values. Many commercial database applications don't impose referential integrity through constraints because companies opt to enforce them through stored programs. A collection of stored programs that protects the integrity of relationships is an application programming interface (API). The benefit of an API is that it eliminates the overhead imposed by foreign key constraints. This also means DML statements run faster without database-level constraint validation.

The downside of foreign key constraints is minimal but important to understand. Although foreign key constraints guarantee referential integrity of data, they do so at the cost of decreased performance. A nice compromise position on foreign keys deploys them in the stage environment (pre-production) to identify any referential integrity problems with your API.



NOTE
A stage environment *is where stable information technology companies conduct end user and final integration testing.*

Foreign key constraints are useful tools for Electronic Data Processing (EDP) auditors regardless of whether they're deployed to maintain referential integrity. For example, an EDP auditor can attempt to add a foreign key constraint to verify whether the API does actually maintain the

Mandatory or Optional Foreign Keys

A mistaken belief among some database developers is that a foreign key constraint restricts a column's cardinality such that it must have a value. A foreign key constraint does not implement a `NOT NULL` constraint. You must assign a `NOT NULL` constraint when you want to prevent the insertion or update of a row without a valid foreign key value.

integrity of relationships. An EDP auditor knows there's a problem with an API when a foreign key can't be added. That type of failure occurs because the data doesn't meet the necessary referential integrity rules. Likewise, an EDP auditor verifies the referential integrity of an API when foreign key constraints can be added to a primary-to-foreign key relationship. Such experimental foreign key constraints are removed at the conclusion of an EDP audit.

There are differences between foreign keys in Oracle or MySQL. The next sections discuss those differences and how to discover foreign key constraints.

Oracle Foreign Key Constraints

An Oracle foreign key constraint is very robust and can have three possible implementations:

- The default implementation prevents the update or deletion of a primary key value when a foreign key holds a copy of that value.
- Another implementation lets you delete the row but not update the primary key column or set of column values. This is accomplished by appending an `ON DELETE CASCADE` clause when creating or modifying the foreign key constraint.
- Another implementation updates the foreign key value to a null value when the row containing the primary key is deleted. Like the other options, you can't update the primary key column value. This implementation doesn't work when a foreign key column has a column-level `NOT NULL` constraint. In that case, any attempt to delete the row holding the primary key raises an ORA-01407 error, which reports that the foreign key column can't be changed to a null value.

You can disable a foreign key constraint in an Oracle database. This would let a `DELETE` statement remove the row that has a primary key value with dependent foreign key values. Enabling the foreign key constraint after deleting the row with the primary key raises an ORA-02298 error. The error indicates that the database can't validate the rule that the constraint supports, which is that every foreign key value must be found in the list of primary keys.

The Oracle database also requires that you add foreign key constraints as out-of-line constraints when creating a table. This means that foreign key constraints are treated as table constraints. You can add self-referencing foreign key constraints during table creation, but inserting values requires that the first row insertion validates its foreign key against the primary key value in the same row. This means the first row must have the same value for the primary and foreign key column or set of columns.

You can also find foreign keys in the `ALL_`, `DBA_`, or `USER_CONSTRAINTS` and `USER_CONS_COLUMNS` administrative views.

```
COL constraint_source FORMAT A38 HEADING "Constraint Name:| Table.Column"
COL references_column FORMAT A38 HEADING "References:| Table.Column"

SELECT   uc.constraint_name||CHR(10)
||      '('||ucc1.TABLE_NAME||'.'||ucc1.column_name||')' constraint_source
,       'REFERENCES'||CHR(10)
||      '('||ucc2.TABLE_NAME||'.'||ucc2.column_name||')' references_column
FROM     user_constraints uc
,        user_cons_columns ucc1
,        user_cons_columns ucc2
WHERE    uc.constraint_name = ucc1.constraint_name
```

```
AND      uc.r_constraint_name = ucc2.constraint_name
AND      ucc1.POSITION = ucc2.POSITION
AND      uc.constraint_type = 'R'
ORDER BY ucc1.TABLE_NAME
,        uc.constraint_name;
```

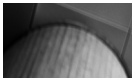
This is similar to the other queries against the database catalog. The only difference is that the constraint type value narrows it to referential integrity.

MySQL Foreign Key Constraints

Foreign key constraints in MySQL require the InnoDB engine, which becomes the default engine beginning with the MySQL 5.5 database. The InnoDB engine is a fully transactional model that supports referential integrity. Other MySQL engines may implement foreign key relationships but don't have the capability of enforcing a database constraint.

MySQL lets you create tables with foreign key constraints. Like the Oracle database, MySQL requires that you use out-of-line or table constraints inside `CREATE TABLE` statements. You also have the option of adding a foreign key constraint after you have created the table. The `ALTER TABLE` statement lets you add a foreign key to any column.

Type mismatches on primary and foreign keys can occur frequently when you're new to MySQL. For example, new users know that a primary key should be a positive number, and they naturally assign an *unsigned integer* as the data type of a surrogate primary key column. The same new users don't know that a foreign key must match exactly so that a foreign key constraint works. This mistake typically involves assigning a signed integer as the data type of a foreign key rather than an unsigned integer.



NOTE
It's a practice in MySQL to use an unsigned integer as the data type for surrogate keys until their sequence values approach the limit of 4.294 billion rows. At that point, you should adopt an unsigned double as the data type.

A type mismatch between a foreign and primary key throws an error when you try to connect, typically an Error 1005. This error message isn't very helpful or actionable, but you can read the InnoDB logs for more information by using the `SHOW ENGINE` command:

```
SHOW ENGINE innodb STATUS;
```

You can see more detail by unfolding the complete log if you're interested in the details. The significant part of the log to solve this type of problem is shown here:

```
------------------------
LATEST FOREIGN KEY ERROR
------------------------
100130 17:16:57 Error IN FOREIGN KEY CONSTRAINT OF TABLE sampledb/#sql-
FOREIGN KEY(member_type)
REFERENCES common_lookup(common_lookup_id):
Cannot find an INDEX IN the referenced TABLE WHERE the
referenced COLUMNS appear AS the FIRST COLUMNS, OR COLUMN types
IN the TABLE AND the referenced TABLE do NOT MATCH FOR CONSTRAINT.
```

You should ensure that all tables use the default InnoDB engine when you want to enforce foreign key relationships by database constraints. Foreign key constraints can be ignored when a designer opts to maintain referential integrity through an API.

You can also discover foreign keys in the `information_schema` database. This query lets you see your foreign keys:

```
SELECT   CONCAT(tc.table_schema,'.',tc.TABLE_NAME,'.',tc.constraint_name)
,        CONCAT(kcu.table_schema,'.',kcu.TABLE_NAME,'.',kcu.column_name)
,        CONCAT(kcu.referenced_table_schema,'.',kcu.referenced_table_name
         ,'.',kcu.referenced_column_name)
FROM     information_schema.table_constraints tc JOIN
         information_schema.key_column_usage kcu
ON       tc.constraint_name = kcu.constraint_name
WHERE    tc.constraint_type = 'FOREIGN KEY'
ORDER BY tc.TABLE_NAME
,        kcu.column_name;
```

CHECK Constraints

`CHECK` constraints let you verify the value of a column during an insert or update. A `CHECK` constraint can set a boundary, such as the value can't be less than, greater than, or between certain values. This differs from the boundary condition imposed by foreign key constraints because `CHECK` constraints qualify their boundaries rather than map them to dynamic values in an external table.

As mentioned earlier in the `NOT NULL` constraint discussion, you can use a `CHECK` constraint to guarantee not null behaviors (qualified as a bad practice). Boundary conditions on the value of a column are typically column-level constraints. You also have set membership conditions. This type of validation works against a set of real numbers, characters, or strings.

Beyond the column-level role of a `CHECK` constraint, there are boundary and set membership conditions where the comparison values are the values of other columns in the same row. When the boundaries are set by the values of other columns in the same row, a `CHECK` constraint becomes a row-level constraint.

A simple boundary or set element example can also apply to row-level constraints. A row-level `CHECK` constraint can disallow the insertion of a null value when another column in the same row would also contain a null value. (A business rule that illustrates this type of need would be a menu item table that has separate columns that classify whether an item belongs on the breakfast, lunch, or dinner menu.)

Beyond boundary and set membership `CHECK` constraints, you have complex business rule conditions that involve checking multiple other columns for sets of business rules. These complex `CHECK` constraints are powerful tools, and in some cases are relegated to database triggers because not all databases implement `CHECK` constraints. Row-level constraints must be implemented in database triggers when `CHECK` constraints aren't supported in a database management system.

Basketball scoring provides a nice business rule for illustrating a *row-level* `CHECK` constraint that is complex. When a player scores a field goal from a shooting position outside the 3-point boundary, the goal is worth 3 points. Any other basket is worth 2 points, unless it is a free throw.

Free throws are worth 1 point. Let's assume the table designed to record points during the game contains the following three columns:

- An optional column (that's null allowed) records whether the basket was made from beyond the 3-point boundary; you enter an X when the condition is met: a field goal.
- An optional column (again, null allowed) records whether a basket was a free throw; you enter an X when the condition is met.
- A mandatory column for the number of points is constrained by values in the optional columns for a field goal and free throws. When the field goal column contains an X, you enter a 3. When the free throw column contains an X, you enter 1. When neither contains an X, you enter 2.

A hidden rule in the foregoing business logic is that an X can be inserted or updated only into the 3-point boundary column when the free throw column is null, and vice versa. It's hidden because it doesn't change the entry of a value for the points scored, only the entry of the Xs in the same row. You would implement `CHECK` constraints on the field goal and free throw columns that would verify that the other was null before allowing the insertion of a value in their respective columns.

The big difference between Oracle and MySQL on `CHECK` constraints is that MySQL doesn't support them. MySQL does support custom data types that mimic the member of sets boundary constraint. The next sections discuss what each database supports.

Oracle CHECK Constraints

Oracle supports boundary constraints, set membership, and complex logic `CHECK` constraints. This means you can avoid writing database triggers for many row-level constraints, which makes implementation actions easier. You will find the basketball example in Chapter 7 where it is shown as part of an `ALTER TABLE` statement.

The query for a `NOT NULL` constraint works for all `CHECK` constraints. You can find the rule enforced by a `CHECK` constraint in the `search_condition` column of the `ALL_`, `DBA_`, or `USER_CONSTRAINTS` view.

MySQL CHECK Constraints

MySQL doesn't support `CHECK` constraints. This means that you must implement row-level constraints inside database triggers. MySQL does support two data types that mimic the set membership condition of `CHECK` constraints: the `ENUM` and `SET` data types.

ENUM Data Type

The `ENUM` data type supports a set membership condition that is like a `CHECK` constraint. When you define a column with an `ENUM` data type, you list the possible elements in the set and any insert or update to the column must be found in the list. What's actually stored in the column is an index value that points to one of the elements in the list. The list is stored as a property of the column in the table definition of the database catalog.

An `ENUM` data type lets you choose one of the values from the list or a null, which makes its default cardinality [0..1] (zero-to-one). You can also place a `NOT NULL` constraint on a column using an `ENUM` data type, which makes its cardinality [1..1] (one-to-one). The `ENUM` data type restricts you to an exclusive choice between available values defined for the column. That means that you can enter the index of only one element from the list of possible values for the `ENUM` data type.

SET Data Type

The `SET` data type also supports a set membership condition and thus mimics a `CHECK` constraint. You are limited to 64 possible values in a `SET` data type. Like the `ENUM` data type, a `SET` data type stores an indexed table of values as a property of the column. A `SET` data type holds a series of strings.

Insertions and updates to a column with a `SET` data type can hold more than one reference index. This makes a `SET` data type an inclusive choice, which means you can enter from 1 to a maximum of 64 elements in a single column. This means that the `SET` data type lets you store an array of potential values in any column that uses the data type.

The default cardinality of a `SET` data type column is [0..64] (zero-to-sixty-four). A `SET` data type column that is `NOT NULL` constrained has a [1..64] (one-to-sixty-four) cardinality.

Trigger Constraints

Database triggers let you implement logic that enforces behaviors like database constraints. They let you restrict column-level and row-level behaviors and allow you to implement external constraints. These external constraints aren't limited to referential integrity through foreign key constraints.

Database triggers also don't let you perform table-level constraints, because they're run after a DML statement begins a transaction against a table. DML statements change the content of tables and fire triggers in the first phase of a two-phase commit (2PC) operation. DML statements also leave the change in an uncommitted state, which means the table is mutating, or undergoing change.

You define triggers to run when an event occurs against a table. Two types of triggers can be defined: statement- and row-level triggers. Triggers have limited options that govern how they act against data in their assigned table. Statement-level triggers can't do anything to the data inside their assigned tables, but row-level triggers can perform the following:

- Column-level validation, substitution, and in validation; that means they can see the proposed change and allow it or substitute values for it.
- Row-level validation, substitution and invalidation; that means they can see all proposed changes to columns and allow them or substitute all or part of them.
- Table-level validation, substitution, and invalidation; that means they can see all proposed changes to columns while exploring values in the same or different tables before making decisions to allow or disallow changes. They can also make substitutions where the values come from other tables.

Triggers are covered in this chapter because they're an option to database constraints. Triggers can be used to add row-level behaviors to any database regardless of whether it supports `CHECK` constraints.

Although triggers work differently in Oracle than they do in MySQL, the different performance characteristics are best left to Chapter 15. You have some foundational knowledge to acquire before you explore the details of how triggers work.

Summary

Database constraints provide you with the ability to restrict the behavior of DML statements at the column, row, and table levels. They also let you enforce external relationship constraints that restrict values from being added, changed, or removed from tables.

This chapter has covered `NOT NULL`, `UNIQUE`, primary key, `CHECK`, and foreign key database constraints, and it provided an introduction to the role of database triggers in restricting or modifying the behavior of DML statements. You have also been exposed to the Oracle and MySQL database implementation differences between these constraints.

Mastery Check

The mastery check is a series of true or false and multiple choice questions that let you confirm how well you understand the material in the chapter. You may check the Appendix for answers to these questions.

1. **True** ☐ **False** ☐ A database constraint can determine whether a column is mandatory or optional.
2. **True** ☐ **False** ☐ A database constraint can determine whether a row is unique in a table.
3. **True** ☐ **False** ☐ A database constraint can only accept or reject values for a column based on boundary conditions.
4. **True** ☐ **False** ☐ You can't add any database constraint during a `CREATE TABLE` statement.
5. **True** ☐ **False** ☐ You can implement referential integrity through database constraints.
6. **True** ☐ **False** ☐ Creating a `UNIQUE` constraint is straightforward and doesn't create any other dependent structures in the database catalog.
7. **True** ☐ **False** ☐ A mutating table doesn't prevent a database trigger from changing the input values of an `INSERT` statement.
8. **True** ☐ **False** ☐ Assigning an `ENUM` data type to a column acts like a `CHECK` constraint.
9. **True** ☐ **False** ☐ MySQL supports referential integrity in the MyISAM database engine.
10. **True** ☐ **False** ☐ A foreign key works when it has a `signed int` data type, while the primary key has an `unsigned int` data type.
11. Which of the following behaviors is/are supported by a primary key constraint (multiple answers possible)?
    - **A.** A column-level behavior
    - **B.** A row-level behavior
    - **C.** A table-level behavior
    - **D.** A mutating table behavior
    - **E.** An external relationship behavior

12. Which of the following behaviors isn't supported by a database trigger?

- **A.** A column-level behavior
- **B.** A row-level behavior
- **C.** A table-level behavior
- **D.** A mutating table behavior
- **E.** An external relationship behavior

13. Which of the following constraints isn't supported by both Oracle and MySQL?

- **A.** A `NOT NULL` constraint
- **B.** A `CHECK` constraint
- **C.** A `UNIQUE` constraint
- **D.** An index constraint
- **E.** A foreign key constraint

14. Which of the following constraints are engine-dependent in MySQL?

- **A.** A `NOT NULL` constraint
- **B.** A `CHECK` constraint
- **C.** A primary key constraint
- **D.** A foreign key constraint
- **E.** A `UNIQUE` constraint

15. What types of conditions aren't supported by `CHECK` constraints?

- **A.** A boundary condition
- **B.** A set membership condition
- **C.** A nullable condition
- **D.** A unique condition
- **E.** A complex business rule based on multiple columns in the same row