MySQL

For Developers

Information Technology Institute

COURSE MATERIALS

You can access the course materials via this link

http://goo.gl/3R3eWh

DAY 2 CONTENTS

- DML
- DRL
- Joins
- Union
- Sub-queries
- Views
- Transactions

INSERTING DATA

- Strings should always be enclosed in pairs of single or double quotation marks in MySQL. Numbers and dates do not need quotes.

```
INSERT INTO table [(column1, column2,
  column3,...)] VALUES

(value1, value2, value3,...);

INSERT INTO table

set column1 = ’ value1’,

column2 = ’ value2’,

column3 = ’ value3’;
```

INSERTING DATA

- The `INSERT...SELECT` syntax is useful for copying rows from an existing table, or (temporarily) storing a result set from a query.

```
INSERT INTO table_name [(column_list)]
  query_expression
```

```
mysql> INSERT INTO Top10_Cities(ID, Name,
  CountryCode)
-> SELECT ID, Name, CountryCode
-> FROM City
-> ORDER BY Population DESC
-> LIMIT 10;
```

INSERTING DATA

- You can use the MySQL-specific function `LAST_INSERT_ID()` to retrieve the first value generated for an `AUTO_INCREMENT` column by the last successful `INSERT` statement.

```
SELECT LAST_INSERT_ID();
```

INSERTING DATA

- In an `INSERT` statement, if you try to insert a row that contains a unique index or primary key value that already exists in the table, you aren't able to add that row. A `REPLACE` statement, however, **DELETES** the old row and **ADDS** the new row

```
REPLACE INTO table

SET column1=value1, column2=value2;
```

INSERTING DATA

- The main difference between `REPLACE` and `ON DUPLICATE KEY UPDATE` is that in case of `REPLACE`, the new row is added to the table, discarding the old row. In the case of `ON DUPLICATE KEY UPDATE`, the old row is preserved, discarding the new row.

```
mysql> INSERT INTO current_users (userid,
username)

-> VALUES (100, 'Tobias')

-> ON DUPLICATE KEY UPDATE visits=visits+1;
```

UPDATING DATA

```
UPDATE [LOW_PRIORITY] [IGNORE] tablename
SET
  column1=expression1,column2=expression2,...
[WHERE condition]
[ORDER BY order_criteria]
[LIMIT number]
```

DELETING DATA

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM
  table
[WHERE condition]
[ORDER BY order_cols]
[LIMIT number]
```

If you write

```
delete from table;
```

on its own, all the rows in a table will be deleted, **so be careful**!

TRUNCATE

```
TRUNCATE [TABLE] table_name;
```

- `TRUNCATE` statement **starts the AUTO_INCREMENT** count over again, unlike the `DELETE` statement.
- `TRUNCATE` is generally **faster** than using a `DELETE` statement as well.
- `TRUNCATE` is transactional statement.

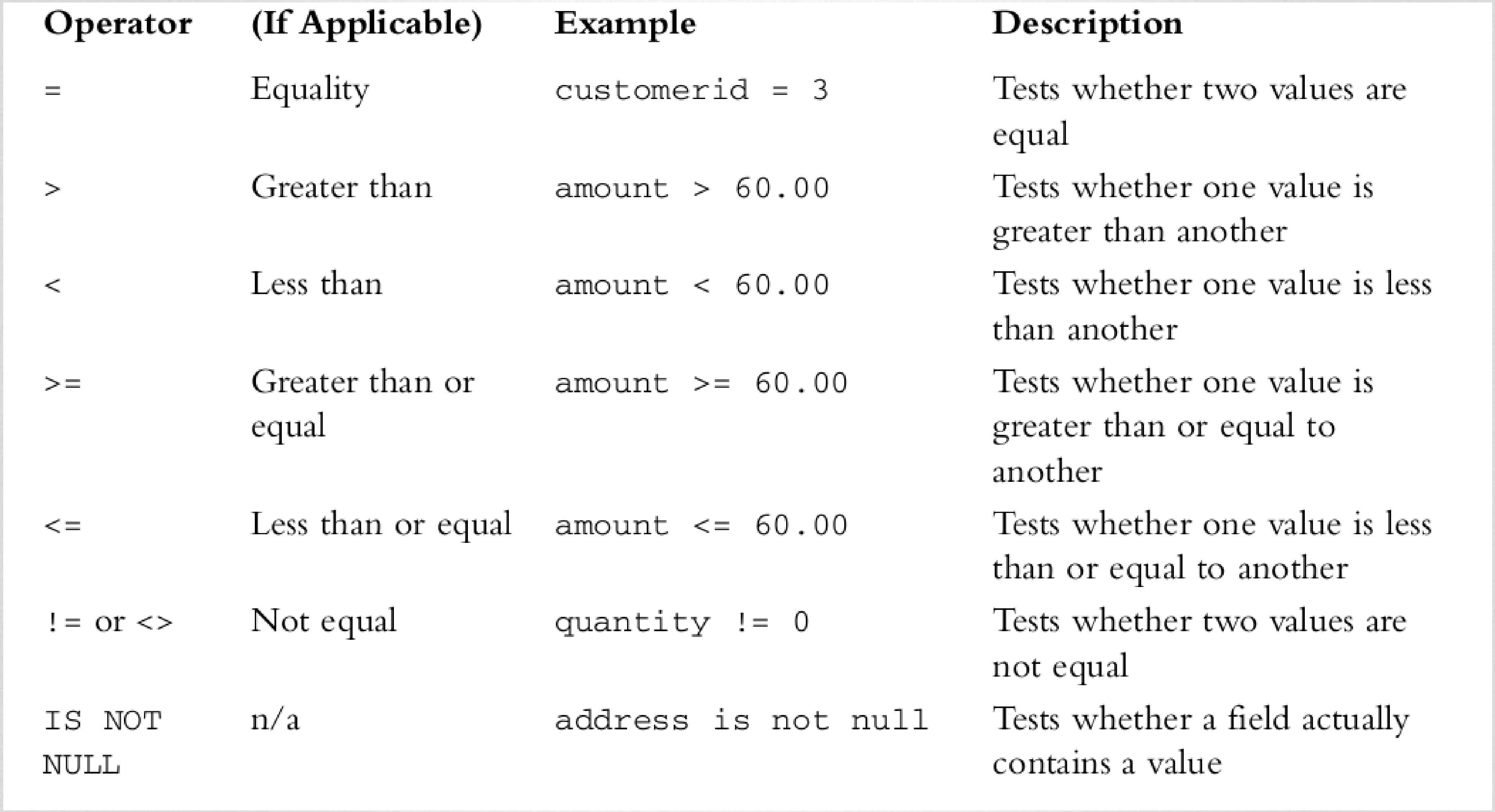
RETRIEVING DATA

- The basic syntax of the `SELECT` statement looks like this

```
SELECT select_list             # What columns to select
FROM table_list                # The tables from which to select rows
WHERE row_constraint           # What conditions rows must satisfy
GROUP BY grouping_columns      # How to group results
ORDER BY sorting_columns       # How to sort results
HAVING group_constraint        # What conditions groups must satisfy
LIMIT count;                   # Row count limit on results
```

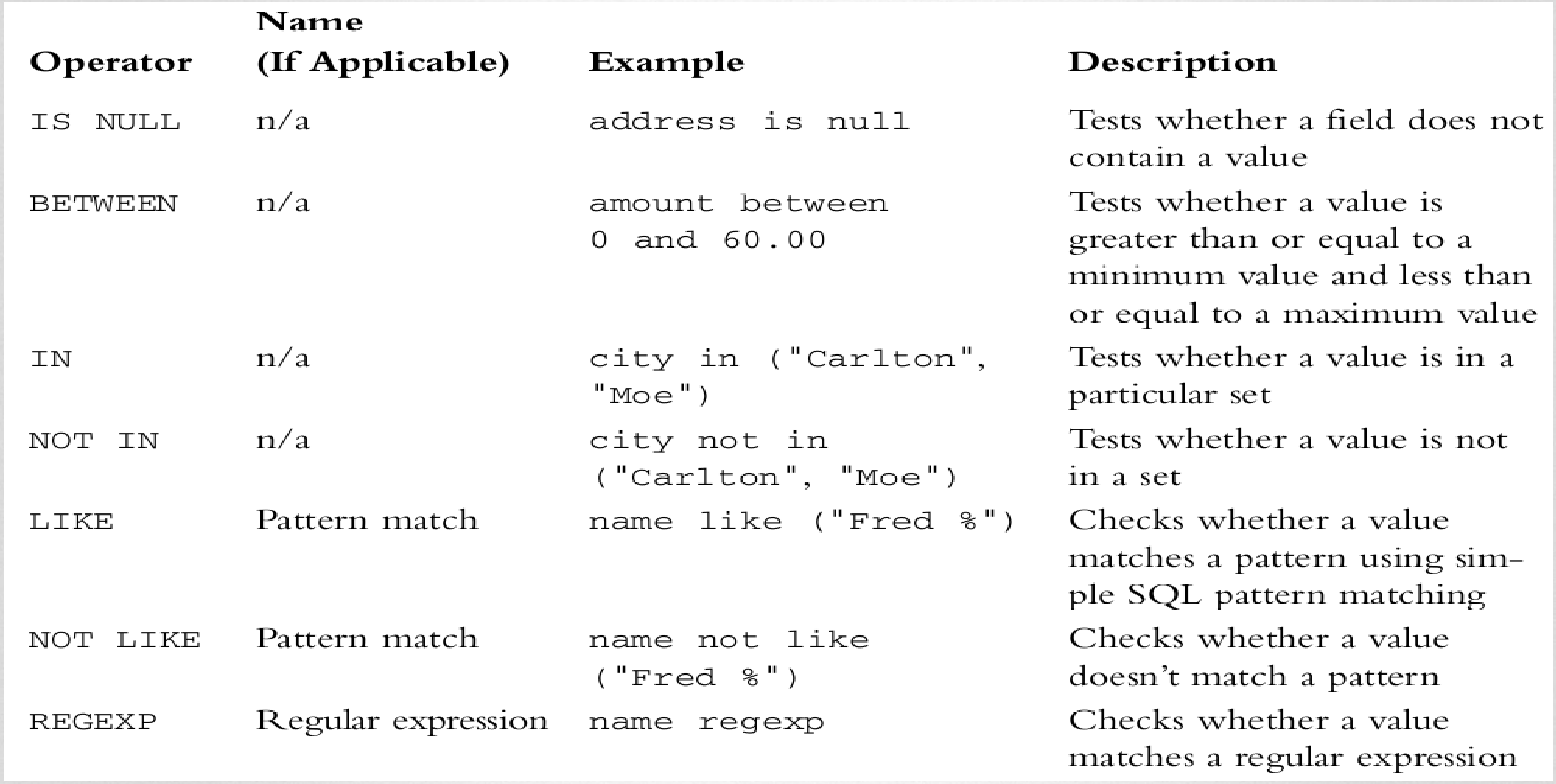
- When the keyword `DISTINCT` is added to the statement, it eliminates the duplicate rows from the result, returning only unique rows.

RETRIEVING DATA



Operator	(If Applicable)	Example	Description
=	Equality	`customerid = 3`	Tests whether two values are equal
>	Greater than	`amount > 60.00`	Tests whether one value is greater than another
<	Less than	`amount < 60.00`	Tests whether one value is less than another
>=	Greater than or equal	`amount >= 60.00`	Tests whether one value is greater than or equal to another
<=	Less than or equal	`amount <= 60.00`	Tests whether one value is less than or equal to another
!= or <>	Not equal	`quantity != 0`	Tests whether two values are not equal
IS NOT NULL	n/a	`address is not null`	Tests whether a field actually contains a value

RETRIEVING DATA



Operator	Name (If Applicable)	Example	Description
`IS NULL`	n/a	`address is null`	Tests whether a field does not contain a value
`BETWEEN`	n/a	`amount between 0 and 60.00`	Tests whether a value is greater than or equal to a minimum value and less than or equal to a maximum value
`IN`	n/a	`city in ("Carlton", "Moe")`	Tests whether a value is in a particular set
`NOT IN`	n/a	`city not in ("Carlton", "Moe")`	Tests whether a value is not in a set
`LIKE`	Pattern match	`name like ("Fred %")`	Checks whether a value matches a pattern using simple SQL pattern matching
`NOT LIKE`	Pattern match	`name not like ("Fred %")`	Checks whether a value doesn't match a pattern
`REGEXP`	Regular expression	`name regexp`	Checks whether a value matches a regular expression

RETRIEVING DATA

- `LIKE` uses simple SQL pattern matching. Patterns can consist of regular text plus the `%` (percent) character to indicate a wildcard match to any number of characters and the _ (underscore) character to wildcard-match a single character.

```
select *

from table

where condition = exp or condition = exp;
```

RETRIEVING DATA

- The `ORDER BY` clause sorts the rows on one or more of the columns listed in the `SELECT` clause. For example,

```
select name, address
from customers
order by name;


order by name asc;
order by name desc;
```

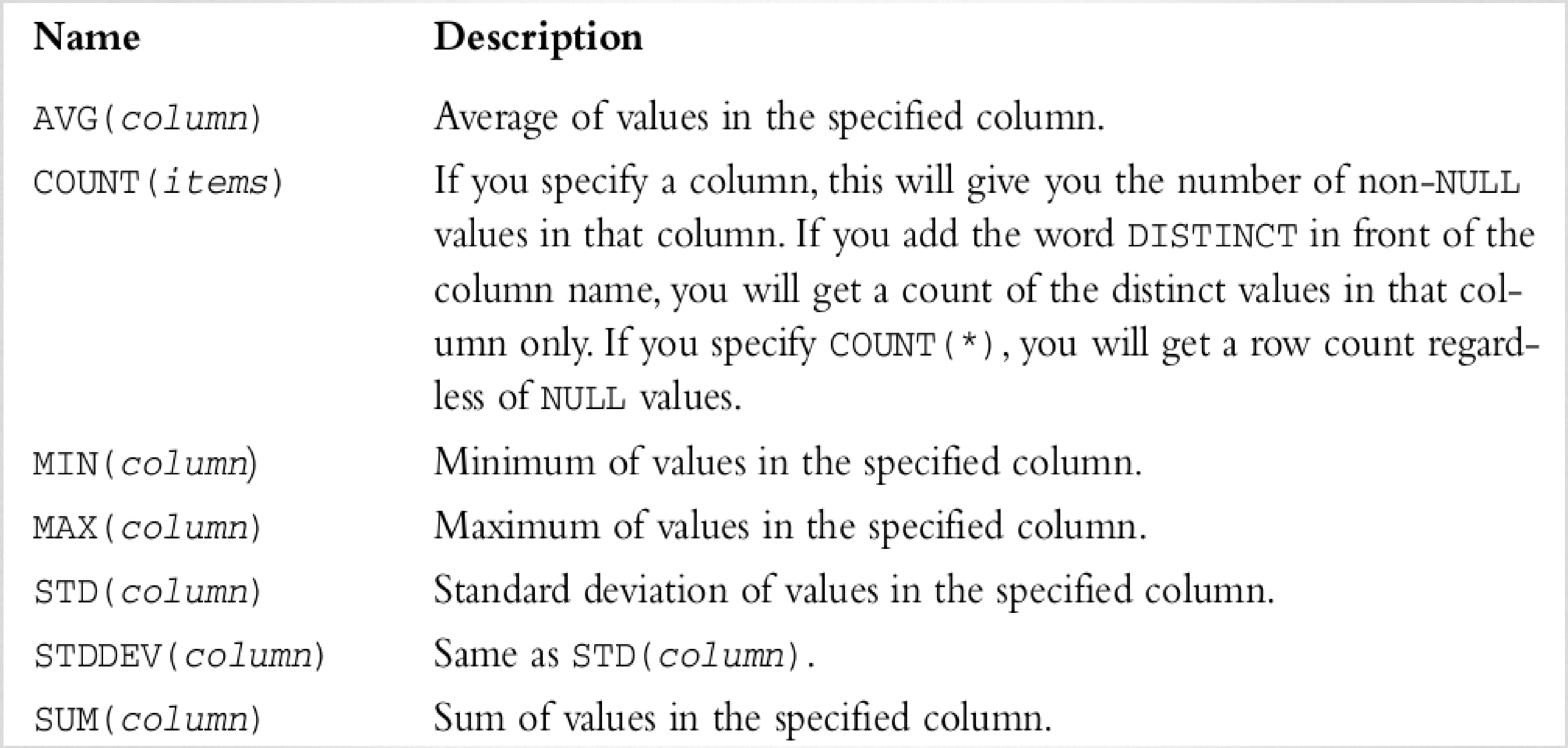
GROUPING AND AGGREGATING DATA

```
select avg(amount)
from orders;
```

- The output is something like this:

```
+--------------+
| avg(amount)  |
+--------------+
|    54.985002 |
+--------------+
```

GROUPING AND AGGREGATING DATA



Name	Description
AVG(*column*)	Average of values in the specified column.
COUNT(*items*)	If you specify a column, this will give you the number of non-NULL values in that column. If you add the word DISTINCT in front of the column name, you will get a count of the distinct values in that column only. If you specify COUNT(*), you will get a row count regardless of NULL values.
MIN(*column*)	Minimum of values in the specified column.
MAX(*column*)	Maximum of values in the specified column.
STD(*column*)	Standard deviation of values in the specified column.
STDDEV(*column*)	Same as STD(*column*).
SUM(*column*)	Sum of values in the specified column.

GROUPING AND AGGREGATING DATA

```
select customerid, avg(amount)
from orders
group by customerid;
```

- When you use a GROUP BY clause with an aggregate function, it actually changes the behavior of the function. Instead of giving an average of the order amounts across the table, this query gives the average order amount for each customer

```
+------------+-------------+
| customerid | avg(amount) |
+------------+-------------+
|          1 |   49.990002 |
|          2 |   74.980003 |
|          3 |   47.485002 |
+------------+-------------+
```

GROUPING AND AGGREGATING DATA

```
select customerid, avg(amount)
from orders
group by customerid
with rollup ;
```

- The WITH ROLLUP modifier can be used in the GROUP BY clause to produce multiple levels of summary values

```
+------------+--------------+
| customerid | avg(amount)  |
+------------+--------------+
|          1 |    49.990002 |
|          2 |    74.980003 |
|          3 |    47.485002 |
|NULL        |   172.455007 |
+------------+--------------+
```

GROUPING AND AGGREGATING DATA

```
select customerid, avg(amount)
from orders
group by customerid
having avg(amount) > 50;
```

- Note that the HAVING clause applies to the groups. This query returns the following output:

```
+------------+--------------+
| customerid | avg(amount)  |
+------------+--------------+
|          2 |    74.980003 |
+------------+--------------+
```

CHOOSING WHICH ROWS TO RETURN

```
select name
from customers
limit 2, 3;
```

- This query can be read as, “Select name from customers, and then return 3 rows, starting from row 2 in the output.” Note that row numbers are zero indexed; that is, the first row in the output is row number zero.

DATA FORM MULTIPLE TABLES

- Simple Two-Table Joins, Ex:

```
select orders.orderid, orders.amount,
  orders.date
from customers, orders
on customers.customerid = orders.customerid;
where customers.name = 'Islam Askar'
```

DATA FROM MULTIPLE TABLES

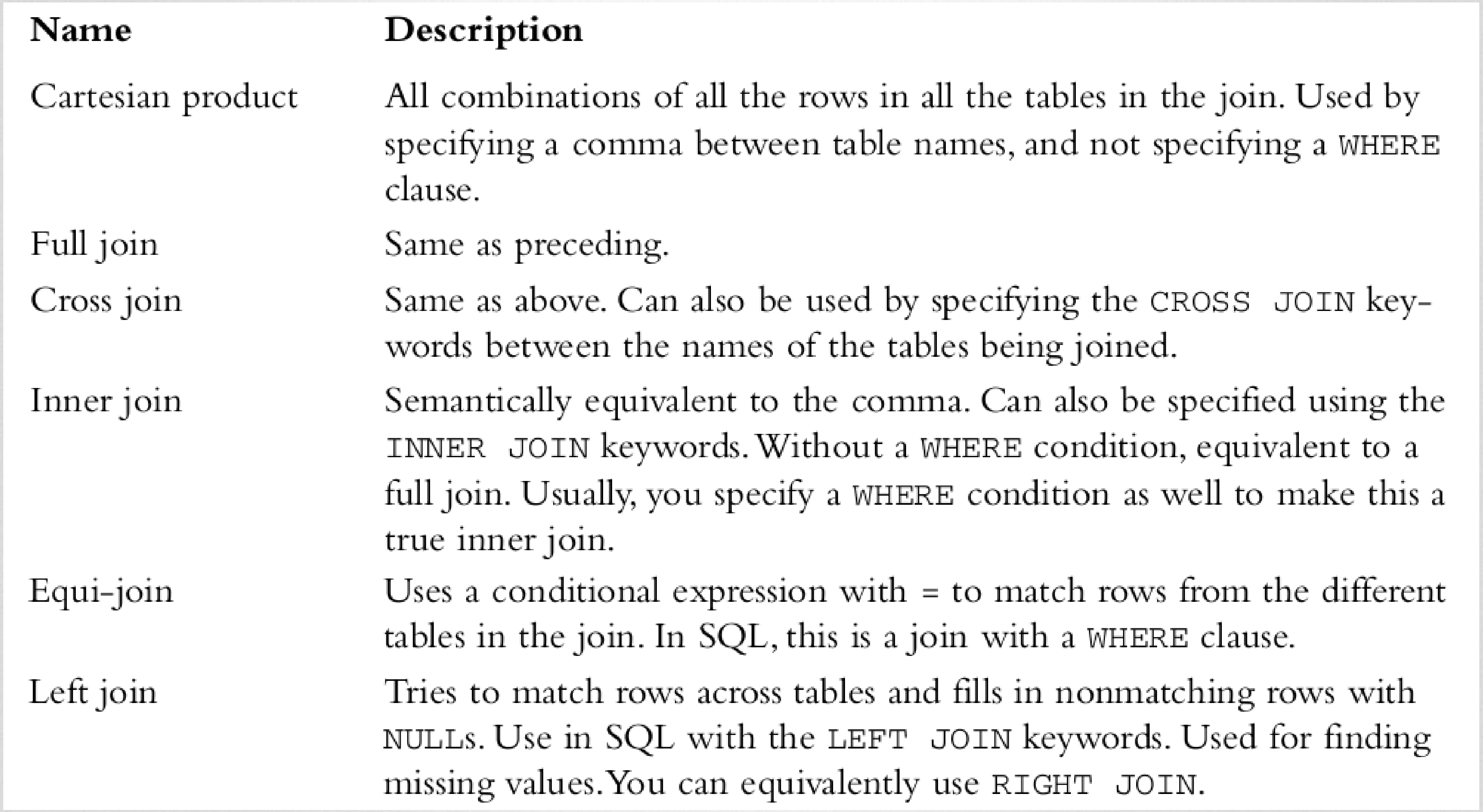
- By listing two tables, you also specify a type of join, possibly without knowing it.
- The `comma` between the names of the tables is equivalent to typing `INNER JOIN` or `CROSS JOIN`. This is a type of join sometimes also referred to as a **full join**, or the **Cartesian product** of the tables.
- By adding this **join condition** to the query, you actually convert the join to a different type, called an **equi-join**.

LEFT & RIGHT OUTER JOIN

- This type of join matches up rows on a specified join condition between two tables. If no matching row exists in the right table, a row will be added to the result that contains `NULL` values in the right columns.
- Let’s look at an example:

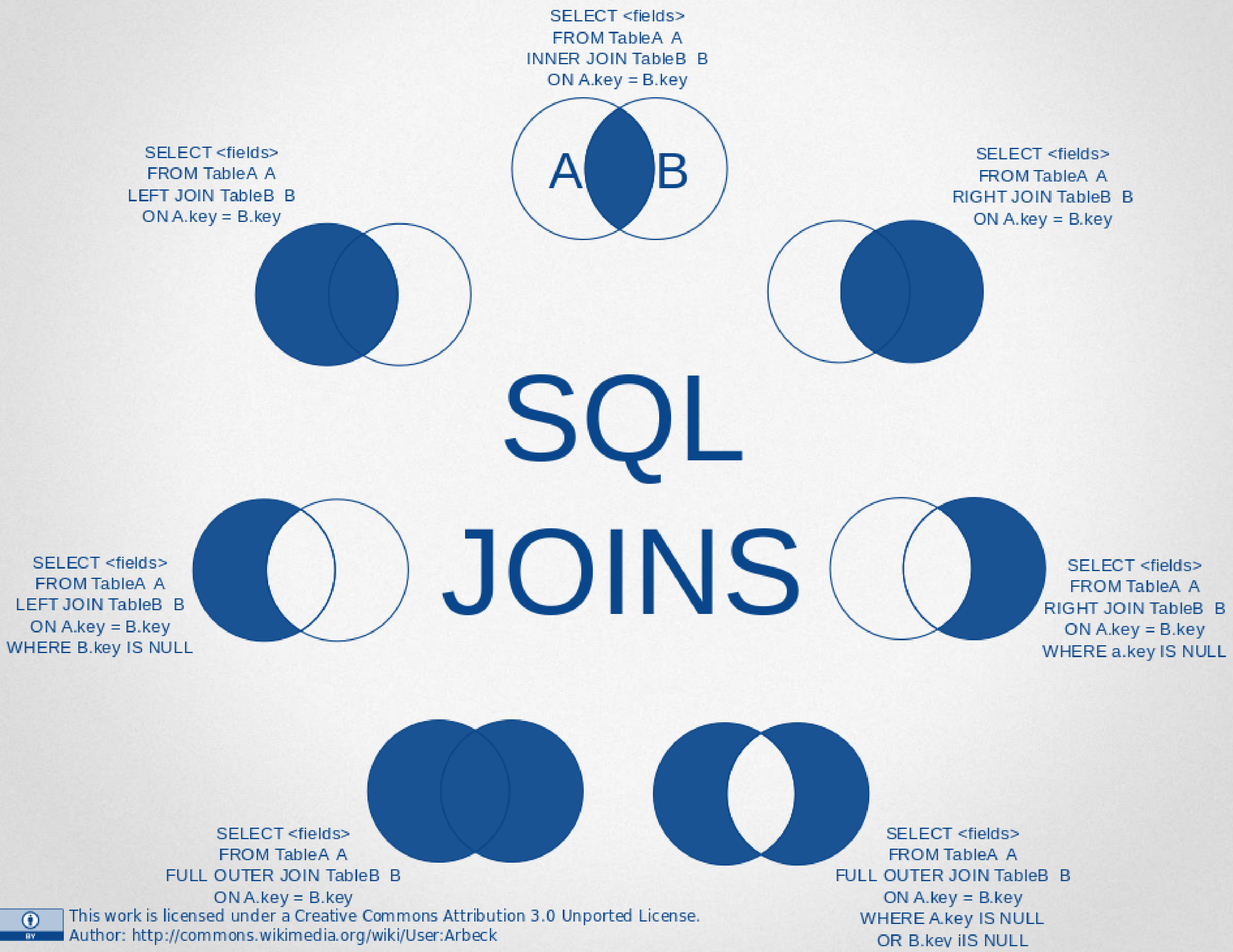
```
select customers.customerid, customers.name,
  orders.orderid
from customers left join orders
on customers.customerid = orders.customerid;
```

SUMMARY OF JOINS



Name	Description
Cartesian product	All combinations of all the rows in all the tables in the join. Used by specifying a comma between table names, and not specifying a `WHERE` clause.
Full join	Same as preceding.
Cross join	Same as above. Can also be used by specifying the `CROSS JOIN` key-words between the names of the tables being joined.
Inner join	Semantically equivalent to the comma. Can also be specified using the `INNER JOIN` keywords. Without a `WHERE` condition, equivalent to a full join. Usually, you specify a `WHERE` condition as well to make this a true inner join.
Equi-join	Uses a conditional expression with = to match rows from the different tables in the join. In SQL, this is a join with a `WHERE` clause.
Left join	Tries to match rows across tables and fills in nonmatching rows with `NULL`s. Use in SQL with the `LEFT JOIN` keywords. Used for finding missing values. You can equivalently use `RIGHT JOIN`.

SUMMARY OF JOINS



```
SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL
```

```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE a.key IS NULL
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key iIS NULL
```

UNIONS

- It is used to combine the results of multiple `SELECT` statements in one result set.
- To join two statements in this way, two main conditions should be obeyed:
  - The statements must return the same number of values
  - The data types for the returned values must correspond to each other

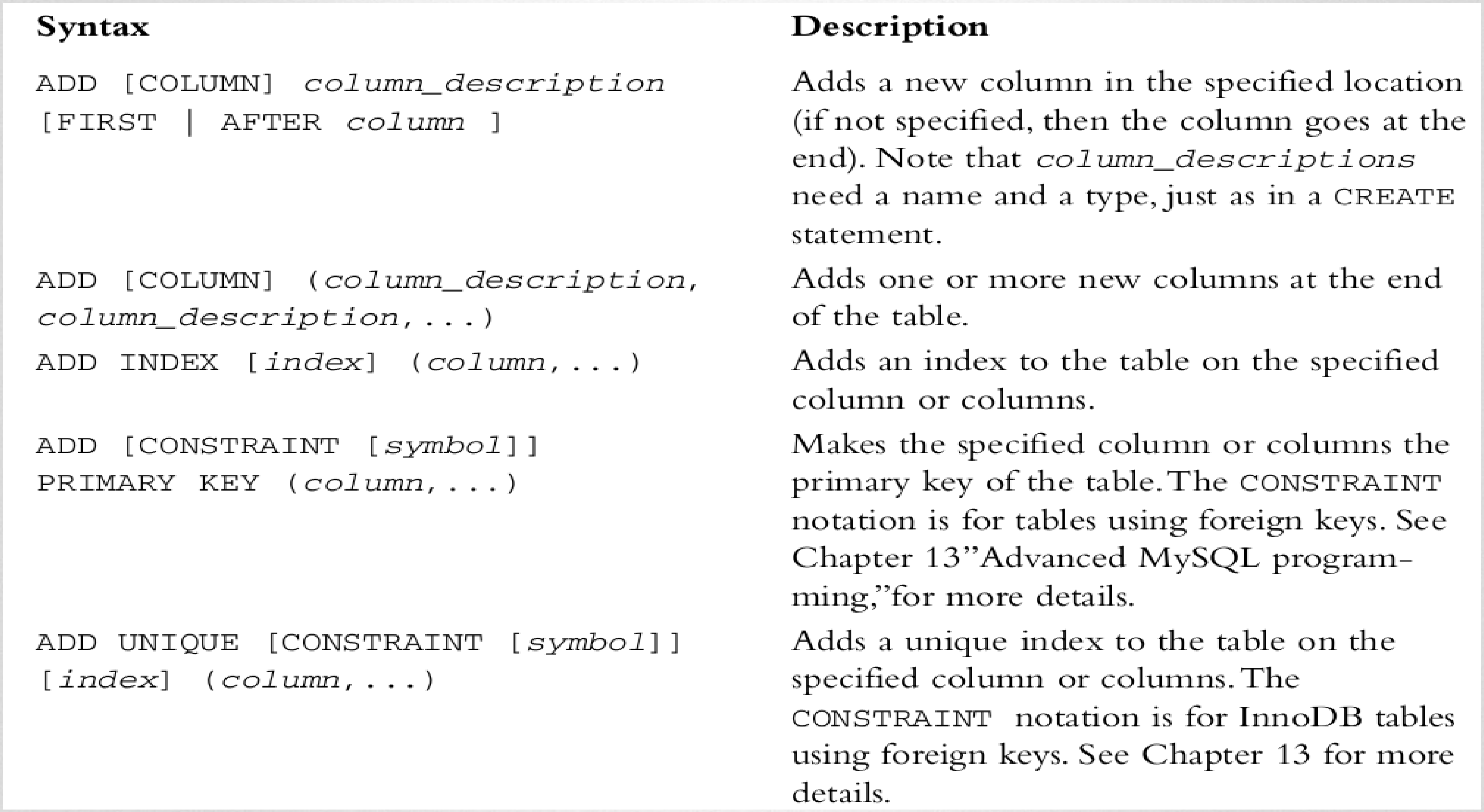
```
SELECT AuthFN, AuthMN, AuthLN
FROM Authors
UNION
SELECT AuthFN, AuthMN, AuthLN
FROM Authors2;
```

ALTERING TABLES AFTER CREATION

The basic form of this statement is:

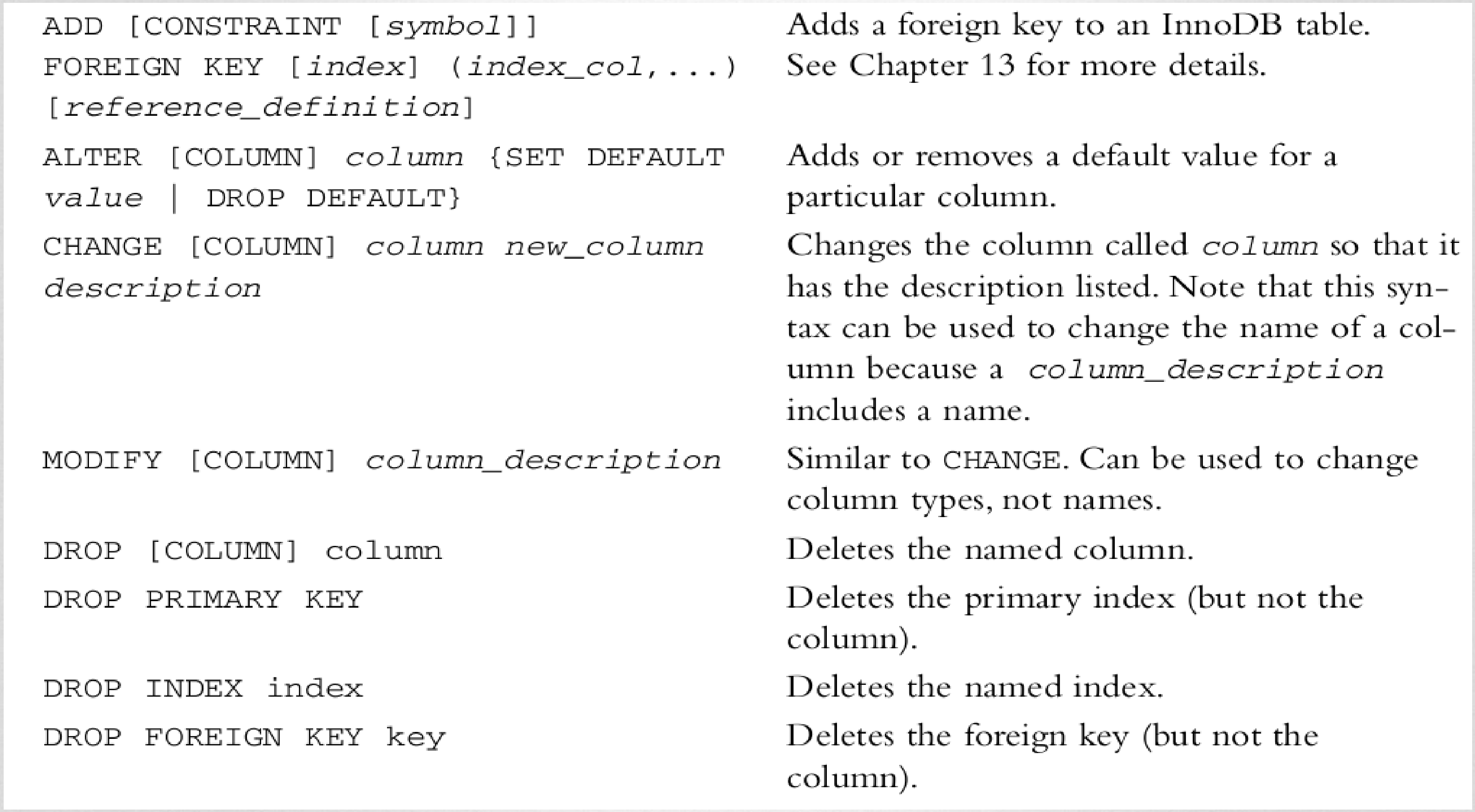
```
ALTER TABLE tablename alteration [,
  alteration ...]

ALTER TABLE tbl_name ENGINE = engine_name;


alter table customers

modify name char(70) not null;
```

ALTERING TABLES AFTER CREATION



Syntax	Description
`ADD [COLUMN] column_description [FIRST | AFTER column ]`	Adds a new column in the specified location (if not specified, then the column goes at the end). Note that `column_descriptions` need a name and a type, just as in a `CREATE` statement.
`ADD [COLUMN] (column_description, column_description,...)`	Adds one or more new columns at the end of the table.
`ADD INDEX [index] (column,...)`	Adds an index to the table on the specified column or columns.
`ADD [CONSTRAINT [symbol]] PRIMARY KEY (column,...)`	Makes the specified column or columns the primary key of the table. The `CONSTRAINT` notation is for tables using foreign keys. See Chapter 13"Advanced MySQL programming,"for more details.
`ADD UNIQUE [CONSTRAINT [symbol]] [index] (column,...)`	Adds a unique index to the table on the specified column or columns. The `CONSTRAINT` notation is for InnoDB tables using foreign keys. See Chapter 13 for more details.

ALTERING TABLES AFTER CREATION



`ADD [CONSTRAINT [`*`symbol`*`]] FOREIGN KEY [`*`index`*`] (`*`index_col`*`,...) [`*`reference_definition`*`]`	Adds a foreign key to an InnoDB table. See Chapter 13 for more details.
`ALTER [COLUMN]` *`column`* `{SET DEFAULT` *`value`* `| DROP DEFAULT}`	Adds or removes a default value for a particular column.
`CHANGE [COLUMN]` *`column new_column description`*	Changes the column called *`column`* so that it has the description listed. Note that this syn-tax can be used to change the name of a col-umn because a *`column_description`* includes a name.
`MODIFY [COLUMN]` *`column_description`*	Similar to `CHANGE`. Can be used to change column types, not names.
`DROP [COLUMN] column`	Deletes the named column.
`DROP PRIMARY KEY`	Deletes the primary index (but not the column).
`DROP INDEX index`	Deletes the named index.
`DROP FOREIGN KEY key`	Deletes the foreign key (but not the column).

RENAMING TABLES

- To rename a table you can use alter:

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

- Also you can use rename:

```
RENAME TABLE old_name TO new_name [,…,…];
```

DROPPING TABLES

- The basic form of this statement to drop table is:

```
DROP TABLE tbl_name;
DROP TABLE IF EXISTS tbl_name;
```

- Drop Temporary table:

```
DROP TEMPORARY TABLE tbl_name;
```

- To Drop multiple tables:

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

SUBQUERIES

- A **subquery** is a query that is nested inside another query.
- The most common use of subqueries is to use the result of one query in a comparison in another query.
- Types of subquery:
  - **Scalar subqueries**: the subquery is treated as single value.
  - **Row subqueries**: the subquery is treated as a single row.
  - **Table subqueries**: the subquery is treated as a (readonly) table that contains zero or more rows

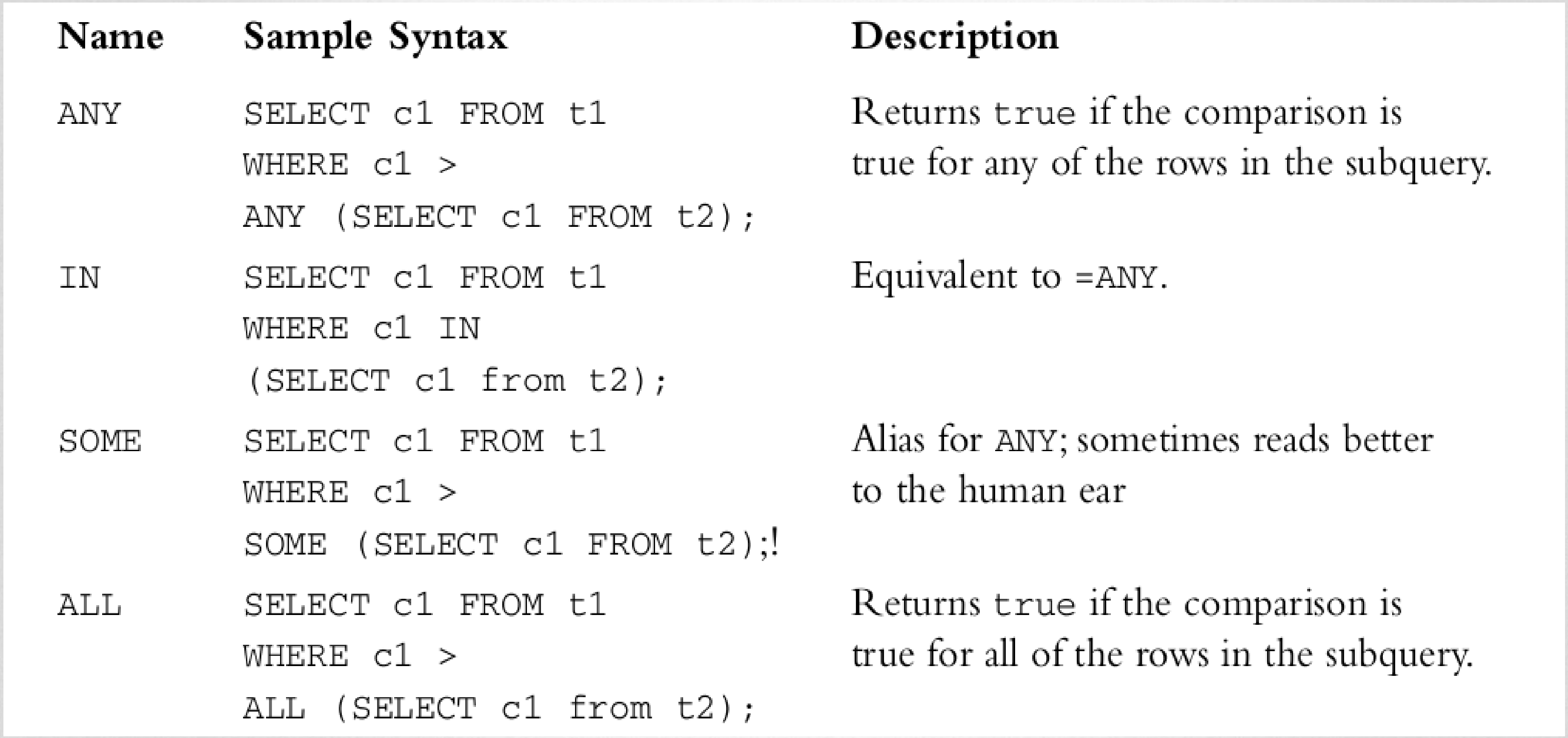
```
select customerid, amount

from orders

where amount = (select max(amount) from
  orders);
```

SCALAR SUBQUERIES

- To evaluate a scalar subquery, its query expression is executed and expected to retrieve at most **one row having exactly one column**.

```
select customerid, amount
from orders
where amount = (select max(amount) from orders);


SELECT Name,
(SELECT COUNT(*) FROM City
WHERE CountryCode = Code) AS Cities,
(SELECT COUNT(*) FROM CountryLanguage
WHERE CountryCode = Code) AS Languages FROM
  Country;
```

TABLE SUBQUERIES



Name	Sample Syntax	Description
ANY	SELECT c1 FROM t1 WHERE c1 > ANY (SELECT c1 FROM t2);	Returns true if the comparison is true for any of the rows in the subquery.
IN	SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 from t2);	Equivalent to =ANY.
SOME	SELECT c1 FROM t1 WHERE c1 > SOME (SELECT c1 FROM t2);!	Alias for ANY; sometimes reads better to the human ear
ALL	SELECT c1 FROM t1 WHERE c1 > ALL (SELECT c1 from t2);	Returns true if the comparison is true for all of the rows in the subquery.

TABLE SUBQUERIES

- Using `FROM` clause

```
SELECT AVG(cont_sum)
FROM   (
SELECT   Continent,
SUM(Population) AS cont_sum
FROM     Country
GROUP BY Continent
) AS t;
```

CORRELATED SUBQUERIES

- In correlated sub-queries, you can use items from the outer query in the inner query.
- For example,

```
select isbn, title

from books

where not exists

(select * from order_items where
  order_items.isbn=books.isbn);
```

VIEWS IN MYSQL

- View is a **virtual** or **logical table** which is composed of result set of a `SELECT` query.
- View is dynamic so it is not related to the physical schema and it is only stored as view definition.
- When the tables which are the source data of a view changes; the data in the view change also.
- In some cases, a view is **updatable** and can be used with statements such as `UPDATE`, `DELETE`, or `INSERT` to modify an underlying base table

VIEWS IN MYSQL

- If the **caching** is enabled the query against view is stored in the cache.
- It increases the performance of query by pulling data from the buffering system instead of making calling to hard disk.
- Queries of views in MySQL are processed in two ways:
  - MySQL creates a temporary table based on the query which defined the view and then execute the input query on this table.
  - MySQL first combines the input query and query which defined the view, then MySQL executes this query.

VIEWS CREATION

- General Form:

```
CREATE [OR REPLACE] [ALGORITHM = {MERGE |
  TEMPTABLE | UNDEFINED}]

VIEW [database_name]. [view_name]

AS [SELECT statement]
```

VIEWS CREATION

- **Algorithms:** The algorithm attribute allows you to control which mechanism is used when creating a view
- **View name:** View name cannot be the same name with existing tables or other views within a database.
- **`SELECT` statement**, you can query any tables or views existed in the database.
- There are several rules which `SELECT` statement has to follow :
  - Temporary tables cannot be used.
  - View cannot be associated with trigger.

EXAMPLE

```
CREATE VIEW vwProducts
AS
SELECT productCode, productName, buyPrice
 FROM products
WHERE buyPrice >
      ( SELECT AVG (buyPrice)
    FROM products )
 ORDER BY buyPrice DESC
```

EXAMPLE

```
CREATE VIEW customerOrders

AS SELECT D.orderNumber, customerName,
SUM(quantityOrdered * priceEach) total

FROM orderDetails D, orders O , customers C

WHERE orderNumber = D.orderNumber

AND

O.customerNumber = C.customerNumber

GROUP BY D.orderNumber ORDER BY total DESC
```

UPDATEABLE VIEWS

- To create **Updateable View:** the `SELECT` statement which defines view has to follow several rules as follow:
  - It must not reference to more than one table. It means it **must not** contain more than **one table** in **FROM** clause, other tables in **JOIN** statement, or **UNION** with other tables.
  - It **must not** use **GROUP BY** or **HAVING** clause.
  - It **must not** use **DISTINCT** in the selection list.
  - It **must not** reference to the **view** that is **not updatable**
  - It **must not** contain any **expression** (aggregates, functions, computed columns…)

UPDATEABLE VIEWS

- Create updatable view:

```
CREATE VIEW officeInfo
AS SELECT officeCode, phone, city
FROM offices
```

- Then you can run update statement

```
UPDATE officeInfo
 SET phone = '+33 14 723 5555'
WHERE officeCode = 4
```

CHANGING VIEWS

- The view can become invalid if a table, view, or column on which it depends is dropped or altered. To check a view for problems of this nature

```
CHECK TABLE view_name\G
```

- To alter a view use:

```
ALTER VIEW view_name AS SELECT * FROM
tbl_name;
```

- To drop a view use:

```
DROP VIEW [IF EXISTS] view_name, [view_name];
```

- TO display the create statement of the view:

```
SHOW CREATE VIEW view_name;
```

VIEWS META-DATA

- You can use these statements also with the view:

```
DESCRIBE view_name;
SHOW COLUMNS FROM view_name;
```

- To show views and tables you can use:

```
SHOW FULL TABLES FROM db_name;
```

VIEWS ADVANTAGES

- Simplify complex query
- Limited access data to the specific users.
- Provide extra security.
- Computed column storing.

TRANSACTIONS

- A transaction is a set of SQL statements that execute as a **unit** and that can be canceled if necessary. Either all the statements execute successfully, or none of them have any effect.
- This is achieved through the use of **commit** and **rollback** capabilities.
- If all of the statements in the transaction succeed, you commit it to record their effects permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was in prior to the point at which the transaction began.

TRANSACTIONS

- The canonical example of this involves a financial transfer where money from one account is placed into another account.
- Suppose that Ahmed writes a check to Islam for $100.00 and Islam cashes the check. Ahmed's account should be decremented by $100.00 and Islam's account incremented by the same amount:

```
UPDATE account SET balance = balance - 100
WHERE name = ‘Ahmed';

UPDATE account SET balance = balance + 100
WHERE name = ‘Islam';
```

TRANSACTIONS

- If a crash occurs between the two statements, the operation is incomplete. Depending on which statement executes first, Ahmed is $100 short without Islam having been credited, or Islam is given $100 without Ahmed having been debited.
- Another use for transactions is to make sure that the rows involved in an operation are not modified by other clients while you’re working with them.
- Transactional systems typically are characterized as providing **ACID** properties. ACID is an acronym for Atomic, Consistent, Isolated, and Durable, referring to four properties that transactions should have:

ACID

- **Atomicity**: The statements transaction consists of form a logical unit. You can’t have just some of them execute.
- **Consistency**: The database is consistent before and after the transaction executes. In other words, the transaction doesn’t make a mess of your database.
- **Isolation**: One transaction has no effect on another.
- **Durability**: When a transaction executes successfully to completion, its effects are recorded permanently in the database.

PERFORMING A TRANSACTION

- To use transactions, you ***must*** use a transactional storage engine such as InnoDB Engines .MyISAM and MEMORY will not work.
- By default, MySQL runs in **autocommit** *mode*, which means that changes made by individual statements are committed to the database immediately to make them permanent.

PERFORMING A TRANSACTION

- To perform transactions explicitly, **disable autocommit** mode and then tell MySQL when to commit or roll back changes.
- One way to perform a transaction is to issue a `START TRANSACTION` (or `BEGIN`) statement to suspend autocommit mode, execute the statements that make up the transaction, and end the transaction with a `COMMIT` statement to make the changes permanent. If an error occurs during the transaction, cancel it by issuing a `ROLLBACK` statement instead to undo the changes.

PERFORMING A TRANSACTION

- The following example illustrates this approach. First, create a table to use

```
CREATE TABLE t (name CHAR(20), UNIQUE (name))
ENGINE = InnoDB;
```

- Then perform the transaction.

```
START TRANSACTION;

INSERT INTO t SET name = ‘Islam';

INSERT INTO t SET name = ‘Ahmed';

COMMIT;

SELECT * FROM t;
```

PERFORMING A TRANSACTION

- Another way to perform transactions is to manipulate the `autocommit` mode directly using `SET` statements:

```
SET autocommit = 0;

SET autocommit = 1;
```

- The effect of any statements that follow becomes part of the current transaction, which you end by issuing a `COMMIT` or `ROLLBACK` statement to commit or cancel it.
- With this method, `autocommit` mode remains off until you turn it back on, so ending one transaction also begins the next one.
- You can also commit a transaction by re-enabling `autocommit` mode.

PERFORMING A TRANSACTION

- To see how this approach works, begin with the same table as for the previous examples:

```
DROP TABLE t;

CREATE TABLE t (name CHAR(20), UNIQUE (name))
ENGINE = InnoDB;
```

- Then disable `autocommit` mode, insert some rows, and commit the transaction:

```
SET autocommit = 0;

INSERT INTO t SET name = 'Islam';

INSERT INTO t SET name = 'Ahmed';

COMMIT;

SELECT * FROM t;
```

TRANSACTIONS AND DDL

- If you issue any of the following statements while a transaction is in progress, the server commits the transaction first before executing the statement:

```
ALTER TABLE

CREATE INDEX

DROP DATABASE

DROP INDEX

DROP TABLE

RENAME TABLE

TRUNCATE TABLE
```

TRANSACTIONS SAVEPOINT

- MySQL enables you to perform a partial rollback of a transaction. To do this, issue a `SAVEPOINT` statement within the transaction to set a marker.To roll back to just that point in the transaction later, use a `ROLLBACK` statement that names the savepoint. The following statements illustrate how this works:

```
CREATE TABLE t (i INT) ENGINE = InnoDB;
START TRANSACTION;
INSERT INTO t VALUES(1);
SAVEPOINT my_savepoint;
INSERT INTO t VALUES(2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO t VALUES(3);
COMMIT;
```