# Ch 6: The Database Language SQL

## 6.1 Simple Queries in SQL

This simple query, like almost all SQL queries, uses the three keywords, ***SELECT***, ***FROM***, and ***WHERE*** that characterize SQL.

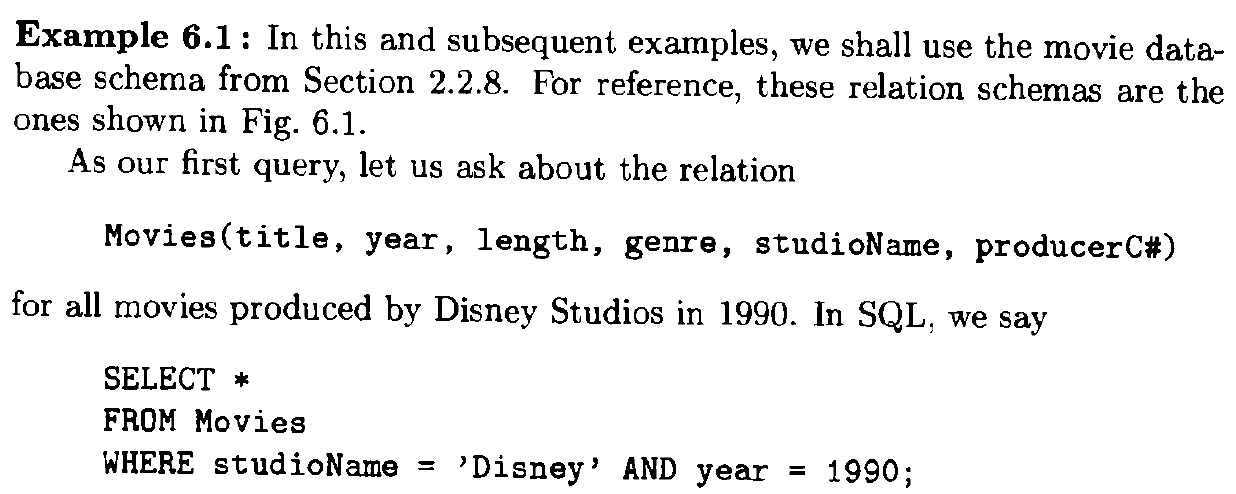
Movies(title, year, length, genre, studioName, producerC#)

Starsin(movieTitle, movieYear, starName)

MovieStar(name, address, gender, birthdate)

MovieExec(name, address, cert#, netWorth)

Studio(name, address, presC#)



* ***FROM*** clause gives the ***relation or relations to which the query refers***.
* ***WHERE*** clause is a ***condition***, much like a selection-condition in relational algebra. ***Tuples must satisfy the condition in order to match the query***.
* ***SELECT*** clause ***tells which attributes of the tuples matching the condition are produced as part of the answer***. The ***\* indicates that the entire tuple is produced***. The result of the query is the relation consisting of all tuples produced by this process.

|  |
| --- |
| **Projection in SQL**  We can project the relation produced by a SQL query onto some of its attributes. In place of the \* of the SELECT clause, we may list some of the attributes of the relation mentioned in the FROM clause. The result will be projected onto the attributes listed.  Sometimes, we wish to ***produce a relation with column headers different from the attributes of the relation mentioned in the FROM clause***. We may follow the name of the attribute by the ***keyword AS*** and an alias, which becomes the header in the result relation. ***Keyword AS is optional***. That is, ***an alias can immediately follow what it stands for, without any intervening punctuation***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 4.46.18 PM.png  Another option in the ***SELECT clause is to use an expression in place of an attribute***. Put another way, the SELECT list can function like the lists in an extended projection.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 4.48.12 PM.png |

|  |
| --- |
| **Selection in SQL**  The selection operator of relational algebra, and much more, is available through the WHERE clause of SQL.  We may build expressions by comparing values using the ***six common comparison operators***: ***=, <> (not equal to), <, >, <=, and >=.***  The values that may be compared include constants and attributes of the relations mentioned after FROM. We may also apply the ***usual arithmetic operators***, +, \*, and so on, to numeric values before we compare them. |

|  |
| --- |
| **Comparison of Strings**  ***Two strings are equal if they are the same sequence of characters***. When comparing strings with different declarations, only the actual strings are compared. When ***we compare strings*** by one of the "***less than" operators***, such as < or >=, we are asking whether one precedes the other in lexicographic order. |

|  |
| --- |
| **Pattern Matching in SQL**  SQL also provides the capability to compare strings on the basis of a simple pattern match.  An ***alternative form of comparison expression*** is  s LIKE p  where ***s is a string*** and ***p is a pattern***, ***a string with the optional use of the two special characters % and \_***. Ordinary characters in p match only themselves in 8. But % in p can match any sequence of 0 or more characters in s, and \_ in p matches any one character in s. The value of this expression is true if and only if string s matches pattern p. Similarly; ***s NOT LIKE p*** ***is true if and only if string s does not match pattern p***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 4.54.49 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 4.54.56 PM.png |

|  |
| --- |
| **Dates and Times**  A ***date constant*** is ***represented*** by the keyword ***DATE*** followed by a ***quoted string*** of a ***special form***. For example, ***DATE '1948-05-14'*** follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month.  The keyword ***TIME*** and a ***quoted string*** ***represent*** a ***time constant*** similarly. This string has two digits for the hour. Then come a colon, two digits for the minute, another colon, and two digits for the second. ***TIME '15: 00:02.5'.***  To ***combine dates and times*** we use a value of type ***TIMESTAMP***. These values consist of the keyword TIMESTAMP, a date value, a space, and a time value. Thus, ***TIMESTAMP '1948-05-14 12:00:00'*** |

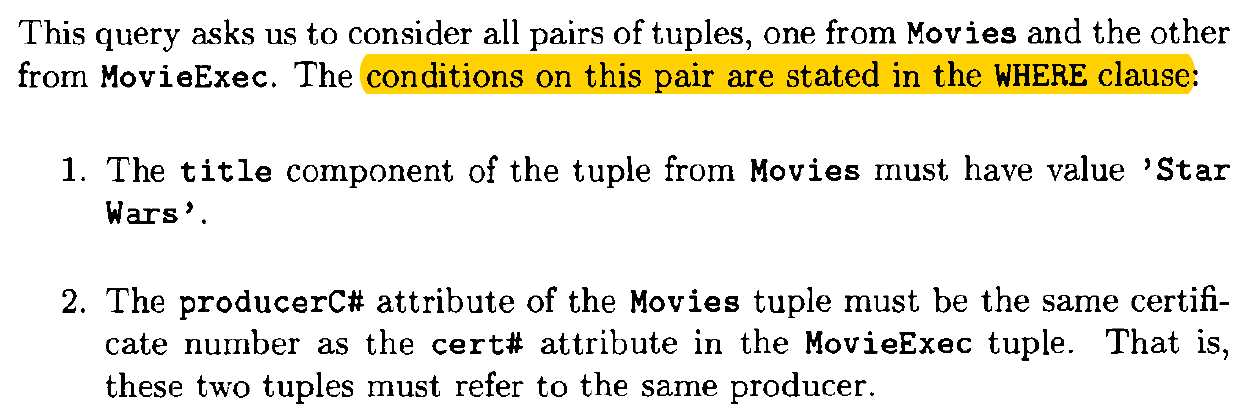
|  |
| --- |
| **Null Values and Comparisons Involving NULL**  ***SQL allows attributes to have a special value NULL***, which is called the null value. There are many different interpretations that can be put on null values.  Here are ***some of the most common***:  1. ***Value unknown***: that is, "I know there is some value that belongs here but I don't know what it is." An unknown birthdate is an example.  2. ***Value inapplicable***: "There is no value that makes sense here." For example, if we had a spouse attribute for the MovieStar relation, then an unmarried star might have NULL for that attribute, not because we don't know the spouse's name, but because there is none.  3. ***Value withheld***: "We are not entitled to know the value that belongs here." For instance, an unlisted phone number might appear as NULL in the component for a phone attribute.  In ***WHERE clauses***, we ***must be prepared*** for the possibility that a component of ***some tuple*** we are examining ***will be NULL***. There are ***two important rules*** to remember ***when we operate upon a NULL value***:  1. When we operate on a NULL and any value, including another NULL, using an ***arithmetic operator*** like x or +, the ***result is NULL***.  2. When we compare a NULL value and any value, including another NULL, using a ***comparison operator*** like = or >, the ***result is UNKNOWN***. The ***value UNKNOWN*** is ***another truth-value, like TRUE and FALSE***. However, we must remember that, although ***NULL*** is a ***value*** that can appear in tuples, it is ***not a constant***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.01.04 PM.png |
| **The Truth-Value UNKNOWN**  The rule is easy to remember if we think of TRUE as 1, FALSE as 0, and UNKNOWN as ½. Then:  1. The ***AND of two truth-values*** is the ***minimum of those values***.  ***x AND y is FALSE if either x or y is FALSE***; it is ***UNKNOWN if neither is FALSE but at least one is UNKNOWN***, and it is ***TRUE only when both x and y are TRUE***.  2. The ***OR of two truth-values*** is the ***maximum of those values***.  ***x OR y is TRUE if either x or y is TRUE***; it is UNKNOWN if ***neither is TRUE but at least one is UNKNOWN, and it is FALSE*** only ***when both are FALSE***.  3. The ***negation of truth-value v is 1- v***. That is, ***NOT x has the value TRUE when x*** is ***FALSE***, the ***value FALSE when x is TRUE*** and the ***value UNKNOWN when x has*** value ***UNKNOWN***. |

|  |
| --- |
| **Ordering the Output**  ***To get output in sorted order***, we may add to the select-from-where statement a clause:  ORDER BY <list of attributes>  The ***order is by default ascending***, but we can get the output highest-first by appending the ***keyword DESC*** (for "descending") to an attribute. The ORDER BY clause follows the WHERE clause and any other clauses.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.06.53 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.06.56 PM.png |

## 6.2 Queries Involving More Than One Relation

SQL has a simple way to couple relations in one query: list each relation in the FROM clause. Then, the SELECT and WHERE clauses can refer to the attributes of any of the relations in the FROM clause.

|  |  |
| --- | --- |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.08.15 PM.png | |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.08.20 PM.png |  |



|  |
| --- |
| **Disambiguating Attributes**  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.10.03 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.10.09 PM.png |

|  |
| --- |
| **Tuple Variables**  ***Disambiguating attributes by prefixing the relation name works as long as the query involves combining several different relations***. However, sometimes we need to ask a query that involves two or more tuples from the same relation. We may list a relation R as many times as we need to in the FROM clause, ***but we need a way to refer to each occurrence of R***. ***SQL allows us to define***, for each occurrence of R in the FROM clause, ***an "alias" which we shall refer to as a tuple variable***. Each use of R in the FROM clause is followed by the (optional) keyword AS and the name of the tuple variable; we shall generally omit the AS in this context. In the ***SELECT and WHERE clauses, we can disambiguate attributes of R by preceding them by the appropriate tuple variable and a dot***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.12.48 PM.png |

|  |
| --- |
| **Interpreting Multirelation Queries**  ***Nested Loops***  The semantics that we have implicitly used in examples so far is that of tuple variables. Recall that a tuple variable ranges over all tuples of the corresponding relation. ***A relation name that is not aliased is also a tuple variable ranging over the relation itself***. ***If there are several tuple variable***s, we ***may imagine nested loops***, one for each tuple variable, i***n which the variables each range over the tuples of their respective relations***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.15.26 PM.png  ***Parallel Assignment***  There is an ***equivalent definition in which we do not explicitly create nested loops ranging over the tuple variables***. Rather, ***we consider in arbitrary order, or in parallel, all possible assignments of tuples from the appropriate relations to the tuple variables***. For each such assignment, we consider ***whether the WHERE clause becomes true***. Each assignment that produces a ***true WHERE clause*** contributes a tuple to the answer; that tuple is constructed from the attributes of the SELECT clause, evaluated according to that assignment.  ***Conversion to Relational Algebra***  A third approach is to ***relate the SQL query to relational algebra***. We start with the tuple variables in the ***FROM clause*** and ***take the Cartesian product of their relations***. ***If two tuple variables refer to the same relation, then this relation appears twice in the product, and we rename its attributes so all attributes have unique names***. |

|  |
| --- |
| **Union, Intersection, and Difference of Queries**  Sometimes we wish to ***combine relations using the set operations of relational algebra: union, intersection, and difference***. SQL provides corresponding operators that apply to the results of queries, provided those queries produce relations with the same list of attributes and attribute types. The ***keywords*** used are ***UNION***, ***INTERSECT***, and ***EXCEPT***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.18.42 PM.png |

## 6.3 Subqueries

In SQL, one query can be used in various ways to help in the evaluation of another. A query that is part of another is called a subquery. Subqueries can have subqueries, and so OIL down as many levels as we wish. There are a number of other ways that subqueries can be used:

1. ***Subqueries can return a single constant***, and this constant can be compared with another value ***in a WHERE clause***.
2. ***Subqueries can return relations*** that can be used in various ways ***in WHERE clauses***.
3. ***Subqueries can appear in FROM clauses***, followed by a tuple variable that represents the tuples in the result of the subquery.

|  |
| --- |
| **Subqueries that Produce Scalar Values**  An ***atomic value that can appear as one component of a tuple*** is referred to as a ***scalar***. A select-from-where expression can produce a relation with any number of attributes in its schema, and there can be any number of tuples in the relation.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.21.09 PM.png |

|  |
| --- |
| **Conditions Involving Relations**  There are a number of SQL operators that we can apply to a relation R and produce a boolean result. However, the relation R must be expressed as a subquery. Some of the operators below- ***IN, ALL, and ANY***- will be explained first in their simple form where a scalar value s is involved.  Here are the definitions of the operators:   1. ***EXISTS R*** is a ***condition that is true if and only if R is not empty***. 2. ***s IN R*** is ***true if and only if s is equal to one of the values in R***. Likewise, ***s NOT IN R is true if and only if s is equal to no value in R***. Here, we assume R is a unary relation.   3. ***s > ALL R*** is ***true if and only if s is greater than every value in unary relation R***. Similarly, the ***> operator could be replaced by any of the other five comparison operators***, with the analogous meaning: s stands in the stated relationship to every tuple in R. For instance, s <> ALL R is the same as s NOT IN R.  4. ***s > ANY R*** is ***true if and only if s is greater than at least one value in unary relation R***. Similarly, any of the other five comparisons could be used in place of >, with the meaning that s stands in the stated relationship to at least one tuple of R. For instance, ***s = ANY R is the same as s IN R***.  The EXISTS, ALL, and ANY operators can be negated by putting NOT in front of the entire expression. |

|  |
| --- |
| **Correlated Subqueries**  The simplest subqueries can be evaluated once and for all, and the result used in a higher-level query. A ***more complicated use of nested subqueries requires the subquery to be evaluated many times***, once for each assignment of a value to some term in the subquery that comes from a tuple variable outside the subquery. A ***subquery of this type is called a correlated subquery***. Let us begin our study with an example.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.25.48 PM.png |

|  |
| --- |
| **Subqueries in FROM Clauses**  Another use for sub queries is as relations in a FROM clause. In a FROM list, instead of a stored relation, we may use a parenthesized subquery.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.27.20 PM.png |

|  |
| --- |
| **SQL Join Expressions**  The ***simplest form of join*** expression is a ***cross join***; that term is a synonym for what we called a Cartesian product. For instance, if we want the product of the two relations:  Movies(title, year, length, genre, studioName, producerC#)  Starsin(movieTitle, movieYear, starName)  we can say  Movies CROSS JOIN Starsin;  and the ***result will be a nine-column relation with all the attributes of Movies and Starsin***. Every pair consisting of one tuple of Movies and one tuple of Starsin will be a tuple of the resulting relation |

|  |
| --- |
| **Natural Joins**  ***A natural join differs from a theta-join in that***:  1. ***The join condition*** is that all ***pairs of attributes from the two relations having a common name are equated***, and there are no other conditions.  2. ***One of each pair of equated attributes is projected out***.  The SQL natural join behaves exactly this way. Keywords ***NATURAL JOIN*** appear  between the relations to express the ⋈ operator.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.30.28 PM.png |
| **Outerjoins**  The ***outerjoin operator*** is a way to a***ugment the result of a join by the dangling tuples***, padded with null values. In SQL, we can specify an outerjoin; NULL is used as the null value.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.30.56 PM.png |

## 6.4 Full-Relation Operations

|  |
| --- |
| **Eliminating Duplicates**  A ***relation***, being a set, ***cannot have more than one way of any given tuple***. When a SQL query creates a new relation, the ***SQL system does not ordinarily eliminate duplicates***. Thus, the ***SQL response to a query may list the same tuple several times***.  One of the meaning of a ***SQL select-from-where query*** is that we ***begin with the Cartesian product of the relations referred to in the FROM clause***. ***Each tuple of the product is tested by the condition in the WHERE clause***, and the ***ones that pass*** the test ***are given to*** the ***output for projection*** according to the SELECT clause. This ***projection may cause the same tuple to result from different tuples*** of the product, and if so, ***each copy of the resulting tuple is printed in its turn***.  If we do ***not wish duplicates*** in the result, then we may ***follow*** the ***keyword SELECT*** ***by the keyword DISTINCT***. That word ***tells SQL to produce only one copy*** of any tuple and is the SQL analog of applying the δ operator to the result of the query.  SELECT DISTINCT name |

|  |
| --- |
| **Duplicates in Unions, Intersections, and Differences**  ***The union intersection, and difference operations normally eliminate duplicates.*** That is, ***bags are converted to sets***, and the ***set version of the operation is applied***. In order to prevent the elimination of duplicates, we must follow the operator UNION, INTERSECT, or EXCEPT by the keyword ALL.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.42.49 PM.png |

|  |
| --- |
| **Grouping and Aggregation in SQL**  SQL provides all the capability of the ϒ operator through the use of aggregation operators in SELECT clauses and a special GROUP BY clause. |

|  |
| --- |
| **Aggregation Operators**  SQL uses the five aggregation operators ***SUM, AVG, MIN. MAX. and COUNT***. These operators are ***used by applying them to a scalar valued expression in a SELECT clause***. One ***exception is the expression COUNT ( \*)***, which ***counts all tuples*** in the relation that is constructed from the FROM clause and WHERE clause of the query.  We can ***eliminate duplicates from the column before*** applying the ***aggregation*** ***operator*** by using the keyword ***DISTINCT***. That is, an expression such as ***COUNT(DISTINCT x)*** ***counts the number of distinct values in column x***.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.51.42 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.51.48 PM.png |
| **Grouping**  To group tuples, we use a ***GROUP BY*** clause, ***following the WHERE clause***. The keywords GROUP BY are followed by a list of grouping attributes. In the simplest situation, there is only one relation reference in the FROM clause, and this relation has its tuples grouped according to their values in the grouping attributes. Whatever aggregation operators are used in the SELECT clause are applied only within groups.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 5.53.00 PM.png  Observe in Example 6.31 how the ***SELECT clause has two kinds of terms***. These are the only terms that may appear when there is an aggregation in the SELECT clause.  1. Aggregations, where an aggregate operator is applied to an attribute or expression involving attributes.  2. Attributes, such as studioName in this example, that appear in the GROUP BY clause. In a SELECT clause that has aggregations, only those attributes that are mentioned in the GROUP BY clause may appear unaggregated in the SELECT clause.  It is also possible to use a ***GROUP BY clause in a query about several relations***. Such a query is interpreted by the following sequence of steps:  1. Evaluate the relation R expressed by the FROM and WHERE clauses. That is, relation R is the Cartesian product of the relations mentioned in the FROM clause, to which the selection of the WHERE clause is applied.  2. Group the tuples of R according to the attributes in the GROUP BY clause.  3. Produce as a result the attributes and aggregations of the SELECT clause, as if the query were about a stored relation R. |

|  |
| --- |
| **Grouping, Aggregation, and Nulls**  When tuples have nulls, there are a few rules we must remember:   * ***value NULL is ignored in any aggregation***. It does not contribute to a sum, average, or count of an attribute, nor can it be the minimum or maximum in its column. For example, COUNT(\*) is always a count of the number of tuples in a relation, but COUNT (A) is the number of tuples with non-NULL values for attribute A. * ***NULL is treated as an ordinary value when forming groups***. That is, we can have a group in which one or more of the grouping attributes are assigned the value NULL. * When ***we perform any aggregation except count over an empty bag of values, the result is NULL***. The count of an empty bag is 0. |

|  |
| --- |
| **HAVING Clauses**  We ***could restrict the tuples prior to grouping in a way that would make undesired groups empty***. Sometimes we ***want to choose our groups based on some aggregate property*** of the group itself. Then we follow the ***GROUP BY*** clause with a ***HAVING clause***. The latter clause consists of the ***keyword HAVING*** followed by a condition about the group.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.10.13 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.10.33 PM.png  There are ***several rules we must remember about HAVING clauses***:   * An aggregation in a HAVING clause applies only to the tuples of the group being tested. * Any attribute of relations in the FROM clause may be aggregated in the HAVING clause, but only those attributes that are in the GROUP BY list may appear unaggregated in the HAVING clause (the same rule as for the SELECT clause). |

## 6.5 Database Modifications

We have focused on the normal SQL query form: the select-from-where statement. There are a number of other statement forms that do not return a result, but rather change the state of the database. In this section, we shall focus on three types of statements that allow us to

1. Insert tuples into a relation.
2. Delete certain tuples from a relation.
3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as modifications.

|  |
| --- |
| **Insertion**  The basic form of insertion statement is:  INSERT INTO R(A1,… ,An) VALUES (vi, … ,vn);  A tuple is created using the value vi for attribute Ai, for i = 1, 2, ... , n. If the list of attributes does not include all attributes of the relation R, then the tuple created has default values for all missing attributes.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.16.13 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.17.25 PM.png |
| **Deletion**  The form of a deletion is  DELETE FROM R WHERE <condition>;  The effect of executing this statement is that every tuple satisfying the condition will be deleted from relation R.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.29.56 PM.png |
| **Updates**  While we might think of both insertions and deletions of tuples as "updates" to the database, an ***update in SQL is a very specific kind of change to the database***: one or more tuples that already exist in the database have some of their components changed. The ***general form of an update statement*** is:  **UPDATE R SET <new-value assignments> WHERE <condition>;**  ***Each new-value assignment is an attribute, an equal sign, and an expression***. If there is more than one assignment, they are separated by commas. The effect of this statement is to find all the tuples in R that satisfy the condition. Each of these tuples is then changed by having the expressions in the assignments evaluated and assigned to the components of the tuple for the corresponding attributes of R.  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2018-02-19 at 6.31.26 PM.png |