Data Modelling and Databases - Chapter 6 (Book)

- Author: Ruben Schenk
- Date: 08.03.2021
- Contact: ruben.schenk@inf.ethz.ch

6. The Database Language SQL

6.1 Simple Queries in SQL

One of the simplest forms of queries in SQL is to ask for those tuples of some relation that satisfy a condition. Such a query is analogous to a selection in relational algebra. this simple query uses the three keywords `SELECT`, `FROM`, and `WHERE` that characterize SQL.

Example: As our first query, let us ask about the relation

```
Movies(title, year, length, genre, studioName, producerNum)
```

for all movies produced by Disney Studios in 1990. In SQL, we say:

```sql
/* Code 6.1: Simple SQL query. */
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

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

6.1.1 Projection in SQL

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.

Example: Suppose we wish to modify the query of Code 6.1 to produce only the movie title and length. We may write:

```sql
/* Code 6.2: Projection in SQL. */
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

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.

Example: We can modify Code 6.2 to produce a relation with attributes `name` and `duration` in place of `title` and `length`.

```sql
/* Code 6.3: Attribute renaming in queries. */
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Another option in the `SELECT` clause is to use an expression in place of an attribute.

Example: Suppose we want outputs as in Code 6.3, but with the length in hours. We might replace the `SELECT` clause of that example with:

```sql
/* Code 6.4: Expressions in SELECT clauses.*/
SELECT title as name, length*0.16667 AS lengthInHours
```

Remark: SQL is *case insensitive*, meaning that it treats upper- and lower-case letters as the same letter. Only inside quotes does SQL make a distinction between upper- and lower-case letters.

6.1.2 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: `=, <>, <, >, <=,` and `>=`. The last four operators are as in C, but `<>` is the SQL sym,bol for "*not equal to*" (!=) and `=` in SQL is equality (==).

We may also apply the usual arithmetic operators `+, *,` and so on, to numeric values before we compare them. We may apply the `concatenation` operator `||` to string. For example `'foo' || 'bar'` has value `'foobar'`.

The simple SQL queries that we have seen so far all have the form:

```sql
/*Code 6.5: SQL Queries and Relational Algebra. */
SELECT L
FROM R
WHERE C
```

in which L is a list of expression, R is a relation, and C is a condition. The meaning of any such expression is the same as that of the relational-algebra expression:

$$\pi_L(\sigma_C(R))$$

6.1.3 Comparison of Strings

Two strings are equal if they are the same sequence of characters. When comparing strings with different declarations (`CHAR` , `VARCHAR` , etc.), only the actual strings are compared. SQL ignores any "pad" characters.

When we compare strings by on of the "less than" operators, such as `<` or `>=` , we are asking whether one precedes the other in lexicographic order.

6.1.4 Pattern Matching in SQL

SQL also provides the capability to compare strings on the basis of a simple pattern match. An alternative from of comparison expression is

```sql
/* Code 6.6: Pattern Matching in SQL. */
s LIKE p
```

where s is a string and p is a `pattern` , that is, a string with the optional use of the two special characters `%` and `_` . Ordinary characters ion p match only themselves in s. But `%` in p can match any sequence of 0 or more characters in s, and `_` in p matches any one character in s.

Similarly, `s NOT LIKE p` is true if and only if string s does not match pattern p.

Example: We remember a movie "Star *something*", and we remember that the something has four letters. What could this movie be? We can retrieve all such names with the query:

```sql
/* Code 6.7: Patter matching. */
SELECT title
FROM Movies
WHERE title LIKE 'Star ____';
```

Remark: The convention taken by SQL is that two consecutive apostrophes in a string represent a single apostrophe and do not end the string. Thus, `''s` in a pattern is matched by a single apostrophe followed by an `s`.

6.1.5 Dates and Times

A `date` constant is represented by the keyword `DATE` followed by a quoted string of a special form. For example, `'1948-05-14'` follows the required form, i.e. is of the form `'YYYY-MM-DD'`.

A `time` constant is represented similarly by the keyword `TIME` and a quoted string. For example, `'15:00:02.5'` is of the required form.

To combine dates and times we use a value of type `TIMESTAMP`. These values consist of the keyword TIMESTAMP, a date value, and a time value. Thus, `TIMESTAMP '1948-05-14 12:00:00'` represents noon on May 14, 1948.

6.1.6 Null Values and Comparisons Involving NULL

SQL allows attributes to have a special value `NULL`, which is called the *null value*. The most common interpretations for the null value are:

- *Value unknown*
- *Value inapplicable*
- *Value withheld*

In `WHERE` clauses, we must be prepared for the possibility that a component of some tuple we are examining may be NULL. We must remember that:

1. When we operate on a `NULL` and any value, including another `NULL`, using an arithmetic operator like $\times$ 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 correct way to ask if x has the value `NULL` is with the expression `x IS NULL`. Similarly, `x IS NOT NULL` has the value `TRUE` unless the value of x is `NULL`.

6.1.7 The Truth-Value UNKNOWN

We have just seen that when NULL values occur, comparisons can yield a third truth-value: `UNKNOWN`. We must now learn how the logical operators behave on combinations of all three truth-values.

The rules is easy if we think of TRUE as 1, FALSE as 0, and UNKNOWN as $1/2$. Then:

- The AND of two truth-values is the minimum of those values.
- The OR of two truth-values is the maximum of those values.
- The negation of truth-value v is $1 - v$.

6.1.8 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` to an attribute. The `ORDER BY` clause follows the `WHERE` clause and any other clauses (i.e. the optional `GROUP BY` and `HAVING` clauses, which are introduced in Section 6.4).

Example: To get the movies listed by length, shortest first, and among movies equal length, alphabetically, we can say:

```
/* Code 6.11: Sorting in SQL. */
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

6.2 Queries Involving More Than One Relation

Much of the power of relational algebra comes from its ability to combine two or more relations through joins, products, unions, intersections, and differences.

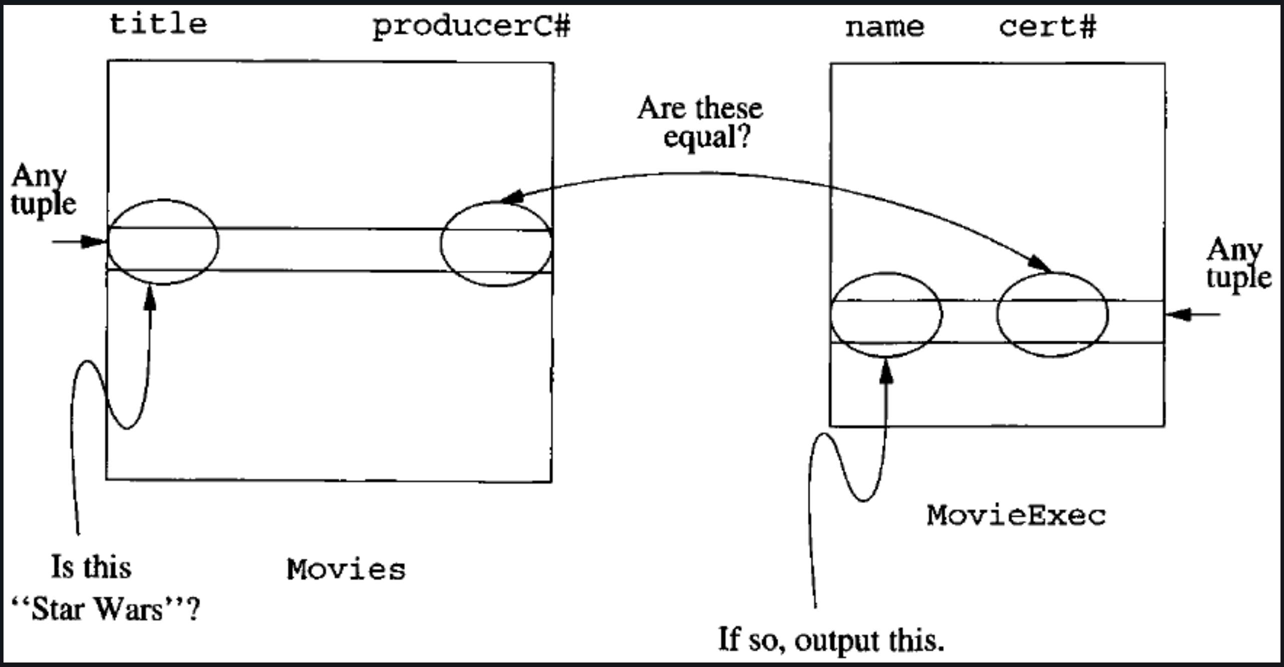
6.2.1 Products and Joins in SQL

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.

Example: Suppose we want to know the name of the producer of *Star Wars*. We can phrase this in one query about the pair of relations *Movies* and *MovieExec* as follows:

```
/* Code 6.12: Coupling relations in SQL. */
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerNum = certNum;
```

The interpretation of the above query is shown in the picture below:



6.2.2 Disambiguating Attributes

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute.

Example: The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, certNum, netWorth)
```

each have attributes *name* and *address*. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job:

```
/* Code 6.13: Relations with the same attributes. */
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

6.2.3 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 lis 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`.

Example: We might want to know about two stars who share an address. The following query does exactly that:

```
/* Code 6.14: Tuple variables. */
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
        AND Star1.name < Star2.name;
```

The second condition in the WHERE clause, *Star1.name < Star2.name*, makes sure that we do produce each star name paired with itself and each pair of stars only once.

6.2.4 Interpreting Multirelation Queries

There are several ways to define the meaning of the select-from-where expressions that we have just covered.

Nested Loops

If there are several tuple variables, we may imagine nested loops, one for each tuple variable, in which the variables each range over the tuples of their respective relations. For each assignment of tuples to the tuple variables, we decide whether the WHERE clause is true.

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.

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.
Having created the product, we apply a selection operator to it by converting the WHERE clause to a selection condition on the obvious way.
Finally, we create from the SELECT clause a list of expressions for a final projection operation.

6.2.5 Union, Intersection, and Difference of Queries

Sometimes we wish to combine relations using the `set operations` of relational algebra: union, intersection, and difference. The keywords used are `UNION`, `INTERSECT`, and `EXCEPT` for $\cup$, $\cap$, and $-$, respectively. Words like UNION are used between two queries, and those queries must be parenthesized.

Example: Suppose we wanted the name and addresses of all female movie stars who are also movie executives with a networth over $ 10'000'000:

```
/* Code 6.16: INTERSECT in SQL queries. */
(SELECT name, address
 FROM MovieStar
 WHERE gender = 'F')
    INTERSECT
(SELECT name, address
 FROM MovieExec
 WHERE netWorth > 10000000);
```

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`. There are a number of 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.

6.3.1 Subqueries that Produce Scalar Values

An atomic value that can appear as one component of a tuple is referred to as a `scalar`.

Sometimes we might deduce from information about keys, of from other information, that there will be only a single value is produced from a select-from-where expression. If so, we can use this expression, surrounded by parentheses, as if it were a constant.

6.3.2 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. As a trick, if we want to apply these operators to a stored table `Foo`, we can use the subquery `(SELECT * FROM Foo)`.

Following are the definitions of the `operators`:

- `EXISTS R` is a condition that is true if and only if R is not empty.
- `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.
- `s > ALL R` is true if and only if s is greater than every value in unary relation R (we might replace `>` with any of the five comparison operators).
- `s > ANY R` is true if and only if s is greater than at least one value in unary relation R (again, we might replace `>` by any comparison operator).

6.3.3 Conditions Involving Tuples

A tuple in SQL is represented by a parenthesized list of scalar values. If a tuple t has the same number off components as a relation R, then it makes sense to compare t and R in expressions of the type listed in the previous section.

Example: A query asking for all the producers of movies in which Harrison Ford stars could look like this:

```
/* Code 6.20: Subqueries. */
SELECT name
FROM MovieExec
WHERE certNum IN
    (SELECT producerNum
     FROM  Movies
     WHERE (title, year) IN
        (SELECT movieTitle, movieYear
         FROM StarsIn
         WHERE starName = 'Harrison Ford')
    );
```

It is important to note that we should analyze any query with *subqueries from the inside out.*

Incidentally, the nested query of *Code 6.20* can, like many nested queries, be written as a single select-from-where expression with relations in the FROM clause for each of the relations mentioned in the main query or a subquery:

```
/* Code 6.20: continuation. */
SELECT name
FROM MovieExec, Movies, StarsIn
WHERE certNum = producerNum AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford'
```

6.3.4 Correlated Subqueries

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.

Example: Consider the following query finding movie titles that appear more than once:

```sql
/* Code 6.21: Correlated subqueries. */
SELECT title
FROM Movies Old
WHERE year < ANY
      (SELECT year
       FROM Movies
       WHERE title = OLd.title);
```

When writing a correlated query it is important that we be aware of the `scoping rules` for names. In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's FROM clause if some tuple variable's relation has that attribute in its schema. If not, we look at the immediately surrounding subquery, then to the one surrounding that, and so on.

However, we can arrange for an attribute to belong to another tuple variable if we prefix it by that tuple variable and a dot. That is why we introduced the alias *Old* for the *Movies* relation of the outer query.

6.3.4 Subqueries in FROM Clauses

Another use for subqueries is as relations in a FROM clause. In a FROM list, instead of a stored relation, we may use a parenthesized subquery. Since we don't have a name for the result of this subquery, we must give it a tuple-variable alias.

Example: Let us reconsider the problem of *Code 6.20*, where we wrote a query that finds the producers of Harrison Ford's movies:

```sql
SELECT name
FROM MovieExec, (SELECT producerNum
                 FROM Movies, StarsIn
                 WHERE title = movieTitle AND
                       year = movieYear AND
                       starName = 'Harrison Ford'
                ) Prod
WHERE certNum = Prod.producerNum;
```

6.3.6 SQL Join Expressions

We can construct relations by a number of variations on the join operator applied to two relations. These variants include products, natural joins, theta-joins, and outerjoins.

Cross Join

The simplest form of join expression is a `cross join`; that term is a synonym for what we called a Cartesian product. We might write:

```
Movies CROSS JOIN StarsIn;
```

Theta-Join

A more conventional `theta-join` is obtained with the keyword `ON`. We put `JOIN` between two relation names R and S and follow them by `ON` and a condition. The meaning of `JOIN...ON` is that the product of $R \times S$ is followed by a selection of whatever condition follows `ON`.

Example: Suppose we want to join the relations *Movies* and *StarsIn* with the condition that the only tuples to be joined are those that refer to the same movie. We can ask this query with:

```
/* Code 6.23: Theta-Join */
Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

6.3.7 Natural Joins

As we recall from Section 2.4.8, 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 $\bowtie$ operator. We might write:

```
MovieStar NATURAL JOIN MovieExec;
```

6.3.8 Outerjoins

The `outerjoin` operator was introduced in a previous section as a way to augment 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.

Example: Suppose we wish to take the outerjoin of the two relations *MovieStar* and *MovieExec*. SQL refers to the standard outerjoin, which pads dangling tuples from both of its argument,a s a *full outerjoin*. The syntax is as follows:

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

If we want `left-` or `right-outerjoin` we add the appropriate word `LEFT` or `RIGHT` in place of `FULL`.

6.4 Full-Relation Operations

In this section we shall study some operations that act on relations as a whole, rather than on tuples individually or in small numbers.

6.4.1 Eliminating Duplicates

SQL's notion of relations differs from the abstract notion of relations presented in Section 2.2. A relation, being a set, cannot have more than one copy of any given tuple. When a SQL query creates a new relation, the SQL system does not ordinarily eliminate duplicates.

Assume we form the Cartesian product of two relations, test each tuple with the WHERE clause and then give the passing tuples 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 once copy of any tuple.

6.4.2 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 versions 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`.

6.4.3 Grouping and Aggregation in SQL

In a previous section we introduced the grouping-and-aggregation operator γ for our extended relational algebra. SQL provides all the capability of the γ operator through he use of `aggregation` operators in SELECT clauses and a special `GROUP BY` clause.

6.4.4 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, typically a column name, in a SELECT clause. One exception is the expression `COUNT(*)`, which counts all the tuples in the constructed relation.

In addition, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword `DISTINCT`, e.g. `COUNT(DISTINCT)` or `SUM(DISTINCT x)`.

6.4.5 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*. Whatever aggregation operators are used in the SELECT clause are applied only within the groups.

Example: The problem of finding the sum of the lengths of all movies for each studio is expressed by:

```sql
/* Code 6.31: Grouping in SQL. */
SELECT studioNAme, SUM(length)
FROM Movies
GROUP BY studioName;
```

6.4.6 Grouping, Aggregation, and Nulls

When the tuples have nulls, there are a few rules we must remember:

- The value NULL is ignored in any aggregation (except for `COUNT(*)`).
- On the other hand, 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.

6.4.7 HAVING Clauses

Sometimes we might want to choose our groups based on some aggregate property of the group itself. The latter clause consists of the keyword `HAVING` followed by a condition about the group.

Example: Suppose we want to print the total film length for only those producers who made at least one film prior to 1930, then we might propose the following query:

```sql
/* Code 6.34: HAVING clauses ion SQL. */
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerNum = certNum
GROUP BY name
HAVING MIN(year) < 1930;
```

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 applies for the SELECT clause)

6.6 Transactions in SQL

6.6.1 Serializability

Example: Assume you want to choose a seat for a flight. Then there might a relation such as:

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

upon which we can issue the query:

```
SELECT seatNo
FROM Flights
 WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatStatus = 'available';
```

When the customer clicks on an empty seat, say 22A, the seat may be reserved with the following query:

```
UPDATE Flights
SET seatStatus = 'occupied'
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25' AND seatNo = '22A';
```

However we run into a problem, when two customers each first issue the first query (both see seat 22A as available) and then one-by-one issue the second query. This results in both reserving the same seat.

The problem is solved in SQL by the notion of a "transaction", which is informally a group of operations that need to be performed together. SQL allows the programmer to state that a certain transaction must be `serializable` with respect to other transactions. That is, these transactions must behave as if they were run `serially` - one at a time, with no overlap.

6.6.2 Atomicity

It is possible for a single operation to put the database in an unacceptable state if there is a hardware or software "crash" while the operation is executing.

Example: Consider transferring CHF 100 from the account numbered 123 to the account 456. We might execute the following steps:

1. Add CHF 10 to account 456 by the SQL update statement:

```sql
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

2. Subtract CHF 100 from account 123 by SQL update statement:

```sql
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

Now, if there is a failure between step 1 and step 2, we have a problem, namely, the bank gave CHF 100 "away".

The problem is that certain combinations of database operations, like the two updates of the previous example, need to be done `atomically`, that is, either they are both done or neither is done.

6.6.3 Transactions

The solution to the problem of serialization and atomicity posed in the previous sections is to group database operations into `transactions`. A transaction is a collection of one or more operations on the database that must be executed atomically.

The SQL command `START TRANSACTION` is used to mark the beginning of a transaction. There are two ways to end a transaction:

1. The SQL statement `COMMIT` causes the transaction to end successfully. Whatever changes to the database were caused by the SQL statement or statements since the current transaction began are installed permanently in the database.
2. The SQL statement `ROLLBACK` causes the transaction to abort. Any changes made in response to the SQL statements of the transaction are undone.

6.6.4 Read-Only Transactions

When a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions. If we tell the SQL execution system that our current transaction is `read-only`, that is, it will never change the database, then it is quite possible that the SQL system will be able to take advantage of that knowledge.

We tell the SQL system that the next transaction is read-only by:

```sql
SET TRANSACTION READ ONLY;
```

We can also inform SQL that the coming transaction may write data by the statement

```
SET TRANSACTION READ WRITE;
```

However, this option is the default.

6.6.5 Dirty Reads

`Dirty data` is a common term for data written by a transaction that has not yet committed. A `dirty read` is a read of dirty data written by another transaction. The risk in reading dirty data is that the transaction that wrote it may eventually abort.

SQL allows us to specify that dirty reads are acceptable for a given transaction. We use the `SET TRANSACTION` statement that we discussed previously. The appropriate from for a transaction like that is:

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

The statement above does two things:

1. The first line declares that the transaction may write data
2. The second line declares that the transmission may run with "isolation level" `read-uncommitted`. That is, the transaction is allowed to read dirty data.

6.6.6 Other Isolation Levels

SQL provides a total of four `isolation levels`. We have already seen serializable and read-uncommitted. The other two are `read-committed` and `repeatable-read`. They can be set by:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

For each, the default is that transactions are read-write, so we can add `READ ONLY` to either statement, if appropriate. Incidentally, we also have the option of specifying:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

However, that is the SQL default and need not be stated explicitly.

The `read-committed` isolation level, as its name implies, forbids the reading of dirty data. However, it does allow a transaction running at this isolation level to issue the same query several times and get different answers, as long as the answer reflects data that has been written by transactions that already committed.

Under `repeatable-read isolation`, if a tuple is retrieved the first time, then we can be sure that the identical tuple will be retrieved again if the query is repeated. However, it is also possible that a second or subsequent execution of the same query will retrieve `phantom tuples`. The latter are tuples that result from insertions into the database while our transaction is executing.

Following is a summary of the differences between the four SQL isolation levels:

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

6.7 Summary of Chapter 6

SQL

The language `SQL` is the principal query language for relational database systems. The most recent full standard is called SQL-99 or SQL3.

Select-From-Where Queries

The most common form of SQL query has the form `select-from-where`. It allows us to take the product of several relations (the `FROM` clause), apply a condition to the tuples of the result (the `WHERE` clause), and produce desired components (the `SELECT` clause).

Subqueries

Select-from-where queries can also be used as `subqueries` within a WHERE clause of `FROM` clause of another query. The operators `EXISTS`, `IN`, `ALL`, and `ANY` may be used to express boolean-valued conditions about the relations that are the result of a subquery in a `WHERE` clause.

Set Operations on Relations

We can take the union, intersection, or difference of relations by connecting the relations, or connecting queries defining the relations, with the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

Join Expressions

SQL has operators such as `NATURAL JOIN` that may be applied to relations, either as queries by themselves or to define relations in a `FROM` clause.

Null Values

SQL provides a special value `NULL` that appears in components of tuples for which no concrete value is available. The arithmetic and logic of `NULL` is unusual. Comparison of any value to `NULL`, even another `NULL`, gives the truth value `UNKNOWN`.

Outerjoins

SQL provides an `OUTER JOIN` operator that joins relations but also includes in the result dangling tuples from one or both relations; the dangling tuples are padded with `NULL`'s in the resulting relation.

The Bag Model of Relations

SQL actually regards relations as bags of tuples, not sets of tuples. We can force elimination of duplicate tuples with the keyword `DISTINCT`, while keyword `ALL` allows the result to be a bag in certain circumstances where bags are not the default.

Aggregations

The values appearing in once column of a relation can be summarized (`aggregated`) by using one of the keywords `SUM`, `AVG`, `MIN`, `MAX`, or `COUNT`. Tuples can be partitioned prior to aggregation with the keywords `GROUP BY`. Certain groups can be eliminated with a clause introduced by the keyword `HAVING`.

Modification Statements

SQL allows us to change the tuples in a relation. We may `INSERT` (add new tuples), `DELETE` (remove tuples), or `UPDATE` (change some of the existing tuples), by writing SQL statements using one of these three keywords.

Transactions

SQL allows the programmer to group SQL statements into `transactions`, which may be committed or rolled back (aborted).

Isolation Levels

SQL defines four `isolation levels` called, from most stringent to least stringent: *serializable* (the transaction must appear to run either completely before or completely after each other transaction), *repeatable-read* (every tuple read in response to a query will reappear if the query is repeated), *read-committed* (only tuples written by transactions that have already committed may be seen by this transaction), and *read-uncommitted* (no constraint on what the transaction may see).