What are two major advantages of using transactions, rather than just executing SQL statements one at a time Explain your two answers briefly.
Here are several advantages of transactions. Question 4 only asks for two of these ACID properties.
#1: Atomicity. All of the transaction is executed, or none of it is. This is important for transactions such as moving money from one account to another.
#2: Isolation. A transaction (running with an isolation level other than Read Uncommitted, such as Serializability) doesn't see partial results of another transaction.
Partial results may reflect work that might be rolled back, or incomplete work, a state of the data that never existed.
#3: Consistency. Operating with serializability, business rules (such as having a business' books balanced across debits, credits and balances) are preserved. That is, if the rules were true before a set of transactions started, and each transaction preserve the rules, then with serializability it's "as if" transactions were executed in some serial order, then at the end of execution of those transactions, the business rules are still true.
#4: Durability. With transactions, you're guaranteed that data that has been committed will not go away, even if there's a failure.

Question 5: SQL uses 3-valued logic, with TRUE, FALSE and UNKNOWN.
Rows don't appear in a query result if the value of the WHERE clause condition is FALSE, and they also don't appear in the result if the value of the WHERE clause is UNKNOWN. Perhaps UNKNOWN could be eliminated from SQL, replaced by FALSE? Nope. Explain why eliminating UNKNOWN that way would change SQL semantics.
Answer 5:
In SQL, the negation of TRUE is FALSE, and the negation of FALSE is TRUE, but the negation of UNKNOWN is still UNKNOWN. If salary is NULL for an employee named Smith, then the value of the WHERE condition "salary > 5" in the query:

```
SELECT name FROM Employees WHERE salary > 5;
```

will be UNKNOWN, and Smith won't be in the result. But for the second query:

```
SELECT name
FROM Employees
WHERE NOT (salary > 5);
```

the value of the WHERE condition "NOT (salary > 5)" is also UNKNOWN, since negation of UNKNOWN is UNKNOWN, so once again Smith won't be in the result.
But if we replaced UNKNOWN with FALSE, the value of "salary > 5" would be FALSE if salary was NULL, and the value of "NOT (salary > 5)" would be TRUE, so for the second query, Smith's name would be in the result, a change to the semantics of SQL.

PRIMARY KEY can never be NULL while UNIQUE can. One Primary key only but **UNIQUE** can have many (multiple unique keys allowed)

COUNT(*) counts tuples where name is NULL.
COUNT(name) only counts tuples where name is not NULL.
COUNT(DISTINCT name) counts duplicate names only once.

Question 15: Write a SQL query whose result is the name of Slopes where at least two different customers did Activities on that slope. Only the name of the slope should appear, and no slope name should appear more than once.

```
SELECT DISTINCT s.sname
FROM Slopes s, Activities a
WHERE s.slope_id = a.slope_id
GROUP BY s.slope_id, s.sname
HAVING COUNT(DISTINCT a.cid) >=2;
```

DELETE does the following:
DELETE Deletes all the tuples from the Employees Table. However, the Employees Table is still there, and tuples can be inserted in it.

DROP does the following:
Drops the Employees Table so that it no longer exists. Not only are the tuples gone, but also since the table is gone, no tuples can be inserted in it.

Question 5:
a) Give one reason why primary keys of relations have indexes.
b) There could be indexes on all attributes of relations,but there aren't. Give one reason why aren't all attributes indexed?
Answer 5a):
Primary keys (and other unique attributes) have to be unique. Having index on primary makes checking uniqueness fast. Searching through entire relation would be much slower. Another reason for indexing primary keys is to speed up queries looking up tuple matching primary key (or range of keys)
Answer 5b):
Indexes take extra space.
Caching indexes takes extra memory.
Updating indexes when relations are modified taking time, slowing
query/transaction execution.

Question 14: Write a SQL query whose result is the age for customers whose level is Beginner and who had an activity on 01/06/09. The result should only include age, and shouldn't include any age more than once.
Answer 14:

```
SELECT DISTINCT age
FROM Customers
WHERE level = 'Beginner'
AND EXISTS
        ( SELECT *
        FROM Activities
        WHERE Activities.cid =Customers.cid
        AND Activities.day = '0/1/06/09' );
```

Question 15: Write a SQL query that determines the total number of activities that took place on each slope. Your result should show the slope-id, the name of the slope and the total number of activities on it. But don't include any slopes that had no activities on them.

```
SELECT Slopes.slope-id, Slopes.sname, COUNT(*)
```

```
FROM Slopes, Activities
WHERE Slopes.slope-id = Activities.slope-id
GROUP BY Slopes.slope-id, Slopes.sname;
```

Question 14: Write a SQL query that outputs the count of the number of different slopes for which the customer named 'Cho' did an activity.

Answer 14:

```
SELECT COUNT(DISTINCT Slopes.slope-id)
FROM Slopes
WHERE EXISTS
        ( SELECT *
        FROM Customers, Activities
        WHERE Customers.cid = Activities.cid
        AND Activities.slope_id = Slopes.slope-id
        AND Customers.cname = 'Cho' );
```

Question 18: What is one advantage, and what is one disadvantage, of having an index on a database table? (Provide only one of each.)

Answer 18:

Advantage: Any one of:

Fast access to database table when query searches on column(s) on which index is defined.

Particular useful for maintaining uniqueness of primary key or other uniqueness constraint without having to search entire table when data is inserted (or unique columns are updated).

Disadvantage: Any one of:

Additional update costs since indexes must be updated when table is modified.

Extra space to store indexes; also extra cache use.

Added complexity of optimization because use of index should be considered.

Question 21: Write a SQL statement that doubles every Monday score in the Scores table.

Answer 21:

```
UPDATE Scores
SET Runs = Runs * 2
WHERE Day = 'Monday';
```