ANSWERS

A11.7 Client server databases and concurrent access

Databases and big data · A level · OCR H446 1.3.2, AQA 7517 4.10.5, Eduqas A500QS 2.5 · about 60 min

What this lesson is about

Many clients, one database: the lost update, and the four ways to prevent it: record locks, serialisation, timestamp ordering and commitment ordering.

- Client server databases
- The lost update
- 1. Record locks
- 2. Serialisation
- 3. Timestamp ordering
- 4. Commitment ordering

Questions

6 marks in all

1. Two clients read a score of 10. One writes 12, then the other writes 13. What is this problem called? [1 mark]

- ☐ **A** A lost update
- ☐ **B** A deadlock
- ☐ **C** A deletion anomaly
- ☐ **D** An orphan record

> **Answer:** A. The second write overwrote the first, so one client's change was lost.

2. Transaction A has locked record X and waits for Y; transaction B has locked Y and waits for X. What is this? [1 mark]

- ☐ **A** Deadlock
- ☐ **B** Serialisation
- ☐ **C** A lost update
- ☐ **D** Timestamp ordering

> **Answer:** A. Each waits for a lock the other holds, so neither can go on without the DBMS stepping in.

3. In timestamp ordering, transaction 3 tries to write a record whose read timestamp is 5. What happens? [1 mark]

- ☐ **A** Transaction 3 is aborted, because a younger transaction has already read the record
- ☐ **B** The write goes ahead and the write timestamp becomes 3
- ☐ **C** Transaction 3 waits for transaction 5 to finish
- ☐ **D** Transaction 5 is aborted

> **Answer:** A. A write is refused if its timestamp is less than the record's read or write timestamp.

4. What does this print? [1 mark]

```
read_ts, write_ts, aborted = {}, {}, set()
schedule = [(1, "read", "x"), (2, "write", "x"), (1, "write", "x"), (3, "read", "x")]
for ts, op, rec in schedule:
    if ts in aborted:
        result = "skipped"
    elif op == "read":
        result = "abort" if ts < write_ts.get(rec, 0) else "ok"
        if result == "ok":
            read_ts[rec] = max(read_ts.get(rec, 0), ts)
    else:
        late = ts < read_ts.get(rec, 0) or ts < write_ts.get(rec, 0)
        result = "abort" if late else "ok"
        if result == "ok":
            write_ts[rec] = ts
    if result == "abort":
        aborted.add(ts)
    print(f"T{ts} {op}: {result}")
```

Answer:

```
T1 read: ok
T2 write: ok
T1 write: abort
T3 read: ok
```

T2 writes after T1's read (2 is not less than 1), then T1's write is too late because T2 has written x.

5. Which are advantages of a client server database over separate copies of the data on each computer? [1 mark]

Tick every answer that is true.

- ☐ **A** Every client sees the same, up-to-date data
- ☐ **B** Integrity rules and access rights are enforced in one place
- ☐ **C** The server can never be a single point of failure
- ☐ **D** Concurrent updates can never cause problems

Answer: A, B. One shared copy is consistent and centrally controlled, but the server can fail and concurrent access still needs control.

6. How does serialisation prevent lost updates? [1 mark]

- ☐ **A** It makes transactions act as if they ran one after another, never overlapping
- ☐ **B** It stores every update as a new timestamped fact
- ☐ **C** It converts the data to JSON before saving
- ☐ **D** It deletes transactions that run at the same time

Answer: A. If no two transactions overlap on the same data, neither can overwrite the other's work.

The task: timestamp ordering

`schedule` is a list of operations in the order they reach the DBMS. Each is `(ts, op, record)`: the transaction's timestamp (a whole number from 1 to 5), `"read"` or `"write"`, and the record's name. Apply timestamp ordering exactly as described above. Every record's read and write timestamps start at 0. For each operation, in order, print `T<ts> <op> <record>: <result>`, where result is: - `skipped` if transaction `ts` has already been aborted (do not restart it; its later operations are skipped); - `abort` if the rules refuse the operation (the transaction is now aborted); - `ok` otherwise, updating the record's timestamp. Then print `committed:` followed by the transactions that were never aborted, and `aborted:` followed by those that were, each as `T<ts>` in increasing order, separated by a comma and a space: for example `aborted: T1, T2, T4`. Fourteen lines in all. The robot does not move.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

# (transaction timestamp, operation, record)
schedule = [
    (1, "read", "balls"),
    (2, "read", "balls"),
    (2, "write", "balls"),
    (1, "write", "balls"),
    (3, "write", "score"),
    (1, "read", "score"),
    (2, "read", "score"),
    (3, "read", "balls"),
    (2, "write", "score"),
    (3, "write", "balls"),
    (5, "write", "led"),
    (4, "write", "led"),
]
```

The hint students can ask for: Keep two dictionaries, the latest read timestamp and the latest write timestamp of each record, both starting at 0. A read is too late if a younger transaction has already written the record; a write is too late if a younger transaction has already read or written it. Keep a set of aborted transactions and check it first.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

# (transaction timestamp, operation, record)
schedule = [
    (1, "read", "balls"),
    (2, "read", "balls"),
    (2, "write", "balls"),
    (1, "write", "balls"),
    (3, "write", "score"),
    (1, "read", "score"),
    (2, "read", "score"),
    (3, "read", "balls"),
    (2, "write", "score"),
    (3, "write", "balls"),
    (5, "write", "led"),
    (4, "write", "led"),
```

```
]

read_ts = {}
write_ts = {}
aborted = set()
seen = set()

for ts, op, record in schedule:
    seen.add(ts)
    if ts in aborted:
        result = "skipped"
    elif op == "read":
        if ts < write_ts.get(record, 0):
            result = "abort"
        else:
            read_ts[record] = max(read_ts.get(record, 0), ts)
            result = "ok"
    else:
        if ts < read_ts.get(record, 0) or ts < write_ts.get(record, 0):
            result = "abort"
        else:
            write_ts[record] = ts
            result = "ok"
    if result == "abort":
        aborted.add(ts)
    print(f"T{ts} {op} {record}: {result}")

print("committed:", ", ".join(f"T{t}" for t in sorted(seen - aborted)))
print("aborted:", ", ".join(f"T{t}" for t in sorted(aborted)))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.