Transactions

Thomas Heinis

t.heinis@imperial.ac.uk
wp.doc.ic.ac.uk/theinis

Imperial College London

Single User Database 1

Alice (her *client* program) issues the following three SQL statements, ***in one go***, to the DBMS:

Example:

1. **update ...**
2. **insert ...**
3. **update ...**

And then between statements 2 and 3 the database "crashes"

Q. But what's a database crash?

Q. What should Alice (her client program) do next?

Single User Database 1

Alice (her *client* program) issues the following three SQL statements, in one go, to the DBMS:

Example:

1. **update** ...
2. **insert** ...
3. **update** ...

And the "database crashes" between statements 2 and 3?

Q. But what's a database "crash"?
It could be that the DBMS (e.g. Oracle, MySQL) crashed. It could be that the computer (the *server*) running the DBMS crashed (power-off, disk-crash). In a network situation, it could be that the connection between the client and server was "lost" and the DBMS/server didn't crash at all!

Q. What should Alice (her client program) do next?
Find out what happened (but how?). Inform Alice. Retry, hoping that the database system has successfully restarted - but from which statement?

Single User Database 2

Alice issues the following three SQL statements, ***in one go***, to the database:

Example:

1. **update** ...
2. **insert** ...
3. **update** ...

And the insert causes a constraint violation

Q. Is this any different from the previous crash scenario?

Q. What would we like to happen if a crash or failure occurs?

Single User Database 2

Alice issues the following three SQL statements, in one go, to the database:

Example:
1. **update** ...
2. **insert** ...
3. **update** ...

And insert causes a constraint violation

Q. Is this any different from the crash scenario?

The client program would be informed much more quickly and hopefully given more precise information on the cause of the violation. If the database computer crashed, its likely that it will take longer to restart the database system (check disks etc) and for the client to reconnect. However, in terms of database consistency, we'd really like to have the same "good" semantics in both cases.

Q. What would we like to happen if a crash or failure occurs?
We'd like the database (and active client programs) to resume or restart in a "consistent" state. Ideally no data should be lost.

Atomicity

Example:

1. **update** ...
2. **insert** ..
3. **update** ...

If we consider the three statements that Alice's program issued, as a *single unit of work* - a **transaction** - then a key property for database systems is **atomicity**, which guarantees *all **or nothing*** execution of the transaction, i.e. on completion of the transaction, the state of the database is that of three statements completed or none.

If a failure occurs (e.g. a computer crash or a constraint violation) during any part of the transaction, then the whole transaction should be aborted and all database modifications done by the transaction **rolled-back**. Immediately, in the case of a constraint violation, or on restart, in the case of a computer crash.

Note: Database atomicity is a property of the DBMS, not of the client program. It's up to client programs to decide what they want to do when such failures occur, e.g. inform the user, re-establish a connection & retry, rollback their own state.

Bank Account Transfer

Consider, transferring £100 from Bob to Alice using the following two SQL statements.

1. **update** accounts **set** balance=balance-100 **where** acct=‘Bob‘;
2. **update** accounts **set** balance=balance+100 **where** acct=‘Alice’

If this sequence wasn’t atomic, then it’s possible that we might remove £100 from Bob’s account but not credit Alice’s account, e.g. a crash occurs after statement 1.

Q. When and what kinds of failures might occur here?

Bank Account Transfer

Consider, transferring £100 from Bob to Alice using the following two SQL statements.

1. **update** accounts **set** balance=balance-100 **where** acct='Bob';
2. **update** accounts **set** balance=balance+100 **where** acct='Alice'

If this sequence wasn't atomic, then it's possible that we might remove £100 from Bob's account but not credit Alice's account, e.g. a crash occurs after statement 1.

Q. When and what kinds of failures might occur here?

- A crash at any time.
- Constraint violation for update 1, and if update 2 was allowed to proceed and succeeded, then Alice would get credited £100 but no-one would be debited!!
- Constraint violation for update 2, and if there was no rollback then Bob would be debited £100 and Alice wouldn't be credited with it.
- Constraint violations aren't the only failures that might happen. We might have expressions that cause overflows, syntax errors, memory full etc.

Multi-user Database

Consider a database with two users, Alice and Bob, who issue the following SQL statements at the same time.

Alice: **select** song, price **from** itunes **where** price<=0.99

Bob: **update** itunes **set** price=price*1.02

Q. What results should Alice see?

Multi-user Database

Consider a database with 2 users, Alice and Bob, who issue the following SQL statements at the same time.

Alice: **select** song, price **from** itunes **where** price<=0.99

Bob: **update** itunes **set** price=price*1.02

Q. What results should Alice see?

All prices before the price rise? All prices after the price rise, or a mixture of the two?

There could be millions of songs.

We (probably) don't want a mixture of the two. Either of the other choices would be acceptable, but this implies that the database system executes the two requests **serially**, first Alice's then Bob's, or vice-versa.

If Bob's **update** starts first it, then Alice might need to wait a very long time.

Airline Seat Allocation

Consider Alice and Bob wanting to choose a seat for a flight. They each issue two SQL statements, first statement **1** and then a few minutes later statement **2**.

1. **select** seat **from** flights **where**
   flight='BA0910' **and** date='2011-02-25' **and** status='available'

2. **update** flights **set** status='occupied' **where**
   flight='BA0910' **and** date='2011-02-25 **and** seat='1A'

```
Time 14:25 Alice issues 1.       Seat 1A included in result
Time 14:26 Bob   issues 1.       Seat 1A included in result
Time 14:29 Bob   issues 2 and gets seat 1A
Time 14:30 Alice issues 2 and gets seat 1A !!!
```

In this example, both statements are fine, but we end up with the person issuing the second update getting the seat or maybe both of them getting it!

Airline Seat Allocation

You might think that the client program could do a test before the update to check if the seat was still available before updating. But what if the the seat was available, and in the milliseconds before the update was issued, someone else grabbed the seat?

We could add a passenger attribute and do something like this:

```
2. update flights set status=‘occupied’, passenger=‘Bob’ where
                flight=‘BA0910‘ and date=‘2011-02-25 and seat=’1A‘ and
                status=‘available’

3. select seat from flights where
                flight=‘BA0910’ and date=‘2011-02-25’ and seat=‘1A’ and
                status=‘occupied‘ and passenger=‘Bob’
```

Now Bob can check if he got the seat or not (1 tuple returned by select). If he got the seat, he would effectively lock out Alice. Alice’s update would not satisfy status=‘available’ and her select would return an empty relation.

Assumes individual SQL statements are executed atomically. What if they’re not?

Transactions

In practice, real scenarios are often more complex and more subtle than what we've seen. It's important that databases provide programmers with guarantees about the consistency of the database in the presence of failures while supporting concurrent actions. Transactions are sequences of database operations that are executed in a consistency-preserving and reliable way and provide the following properties, known as the ACID properties:

Atomicity - All or nothing execution. If any part of a transaction fails, the whole transaction fails and the DBMS will rollback all changes made by the transaction.

Consistency - Transactions do not leave the database in an *inconsistent* state.

Isolation - Concurrent transactions are executed as if no other transaction is running, essentially emulating serialised execution of the transactions.

Durability - The results of a committed transaction are not lost. The DBMS provides mechanisms to recover from system crashes (hardware and software).

Consistency

Transactions must ensure database consistency. That is, a successful transaction, a so-called **committed** transaction, must leave the database in a consistent state. Of course, the database should be consistent before the transaction started.

Database consistency is up to the programmer to ensure. SQL helps by including a variety of integrity constraints like primary and foreign key constraints, not nulls, uniques, checks and assertions that the DBMS will check.

More complex rules that cannot be expressed in SQL, must be checked by the application, although a DBMS can help with features like triggers and stored procedures (not covered in this course).

Atomicity and Durability

If a transaction is aborted, then the DBMS must *rollback* any data and schema modifications (e.g. inserts, updates, deletions) carried out by the transaction.

Rollback is done by a **recovery system** that maintains a log of all changes, including old values, which are used to restore previous states.

The recovery system also handles the durability requirement. For example, if the database system crashes shortly after a client program is informed that a transaction committed, it may be the case that the transaction was not fully written to disk.

Typically when a DBMS starts, its recovery system will check its logs to ensure that successful transactions were saved to disk while aborted or unfinished transactions are rolled back.

Compensating transactions

Although recovery for atomicity and durability seems straightforward, real applications are more complicated. Consider the following sequence of actions:

Alice: Requests £100 from her account at an ATM.

DB: Transaction **update** accounts **set** balance=balance-100 **where** acct='Alice' *commits*

DB: Database crashes before transaction succeeded message is sent to ATM.

Alice: Waits a couple of minutes then leaves penniless and annoyed.

In this example, normal recovery isn't sufficient. The system needs to be programmed to carry out a **compensating transaction** to restore £100 back into Alice's account, for example, if the system doesn't receive a later acknowledgement from the ATM that Alice's money was dispensed.

RAID and Replicated servers

To guard against disk failures, it's important to have a copy of the data and recovery data saved on a second disk, ideally in real-time.

A common approach is to use a **RAID** (Redundant Array of Independent/Inexpensive Disks) multi-disk setup that provides extra availability and performance by dividing (*striping*) and replicating (*mirroring*) data across multiple disks. There are many RAID schemes. RAID 5 for example allows the database server to continue running with reduced performance until the failed disk is replaced.

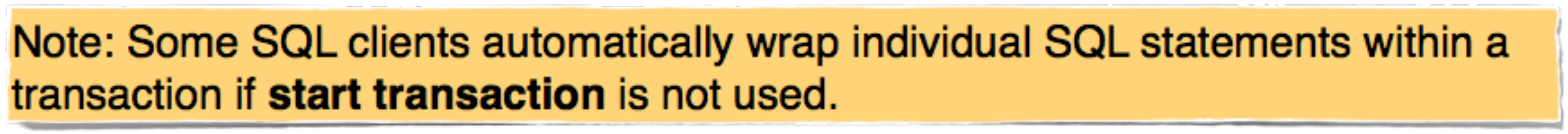
To cope with computer failures we could replicate the database computer, the DBMS and disks; sending database updates to the *replicas*. If the main server computer crashes, then one of the replicas takes over. Another variation on this is to let the replicas also service requests sending updates to each other - such a setup could support more concurrent users but would require support for features such as **distributed transactions**.

Transactions in SQL

SQL is inherently transactional, and ideally a new transaction should start automatically when a transaction commits or rolls back. We can explicitly start a transaction with the **start transaction** statement (**begin** in some systems), commit it with the **commit** statement, or roll it back using the **rollback** statement (of course, rollback will occur automatically for failures etc). We can also set the isolation level for a transaction using the **set transaction** statement.

Example:

```
start transaction;
    select ...;
    insert ...;
    update ...;
commit;
```



Note: Some SQL clients automatically wrap individual SQL statements within a transaction if **start transaction** is not used.

Isolation

Although isolation could be implemented by serialising transactions. In practice, serialised execution is impractical, since it’s important that the resources of the hardware (cpu cores, memory, disks) are fully utilised to support multiple transactions, shorten response times to clients, and increase overall throughput of transactions.

Concurrent transactions are thus an essential feature of *real* databases, but require techniques that preserve database consistency when concurrent transactions access the same data.

Isolation is handled in DBMSs by a **concurrency control system** that allows transactions to execute concurrently as much as possible while ensuring serialisability.

Isolation

Let's consider database concurrency a little bit more. For simplicity we'll consider two concurrent transactions and *we'll assume that database constraints are preserved and consistency maintained*.

Potentially the DBMS could allow both transactions to run concurrently for each of the following cases:

Both transactions are read only. For example, execute select statements only.

The transactions perform modifications, but on different relations.

The transactions perform modifications on the same relations but on different tuples.

Isolation Levels

But what if one transaction modifies a tuple and another wants to access it? In this case we would need to abort or delay one transaction or relax isolation (and hence violate ACID), by allowing changes made by one transaction to become visible to other concurrent transactions. SQL supports the following **isolation levels**:

Read uncommitted. Uncommitted data changed by other concurrent transactions can be read (**dirty reads**). This is the lowest isolation level in SQL, essentially all changes by other transactions are immediately visible to this transaction.

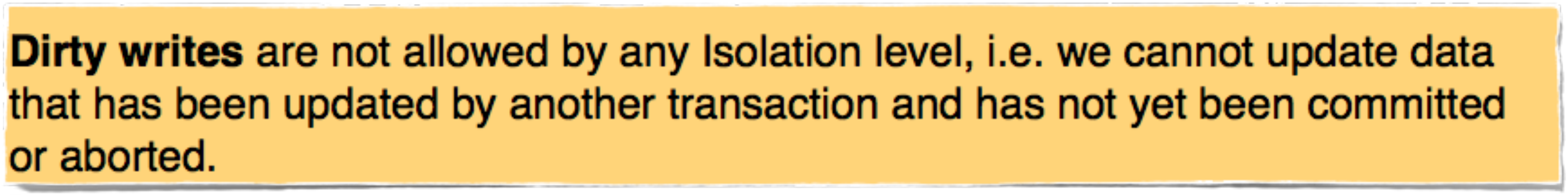
Read committed. Only data committed by other concurrent transactions can be read.

Repeatable read. Like read committed, but guarantees that all tuples returned by a query will be *included* if the query is repeated. The repeated query might return additional newly committed tuples however (**phantom reads**), e.g. a range query might return newly inserted data that falls within the range.

Serializable. Guarantees isolation, but may limit the degree of concurrency.

Isolation Level Anomalies

ISOLATION LEVEL	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read Uncommitted	***Possible***	***Possible***	***Possible***
Read Committed	Prevented	***Possible***	***Possible***
Repeatable Read	Prevented	Prevented	***Possible***
Serializable	Prevented	Prevented	Prevented



Dirty writes are not allowed by any Isolation level, i.e. we cannot update data that has been updated by another transaction and has not yet been committed or aborted.

Imperial College London

name	bal
Alice	10
Bob	20

Dirty Read

During a transaction a tuple is read twice but attribute values differ.

Start Transaction 1	Start Transaction 2
select bal **from** accounts **where** name='Alice'	
	update accounts **set** bal=99 **where** name='Alice'
select bal **from** accounts **where** name='Alice'	
	rollback

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	Prevented	Prevented	***Possible***

In Serializable, Repeatable Read and Read Committed, the 2nd select must return (Alice,10), so the DBMS will lock Alice (the tuple) and cause the update to block until transaction 1 commits. For Read Uncommitted, the 2nd select will return (Alice,99). However transaction 2 could subsequently rollback!

Non-Repeatable Read

name	bal
Alice	10
Bob	20

During a transaction a tuple is read twice but attribute values differ.

Start Transaction 1	Start Transaction 2
select * **from** accounts **where** name='Alice'	
	update accounts **set** bal=99 **where** name='Alice' **commit**
select * **from** accounts **where** name='Alice' **commit**	

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	Prevented	***Possible***	***Possible***

In Serializable and Repeatable Read, the 2nd **select** must return (Alice,10), so will lock Alice and cause the **update** to block until transaction 1 commits. For the other isolation levels, the 2nd **select** will return (Alice,99).

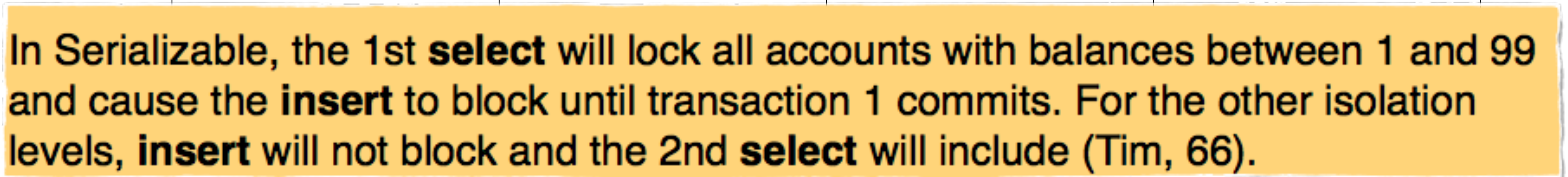
Phantom Read

name	bal
Alice	10
Bob	20

Special case of non-repeatable reads. Tuples returned by a second identical query differ from tuples returned from first query.

Start Transaction 1	Start Transaction 2
select * **from** accounts **where** bal>0 **and** bal<100	
	insert into accounts **set** values(Tim, 66)
	commit
select * **from** accounts **where** bal>0 **and** bal<100	

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	***Possible***	***Possible***	***Possible***


In Serializable, the 1st **select** will lock all accounts with balances between 1 and 99 and cause the **insert** to block until transaction 1 commits. For the other isolation levels, **insert** will not block and the 2nd **select** will include (Tim, 66).

Final Remarks

Transactions are one of the most important concepts in computing. In relational databases they are a fundamental requirement. The key properties that are supported are the ACID properties: atomicity, consistency, isolation and durability.

Atomicity and durability are implemented using a recovery system that logs changes to stable storage and uses the log to restore a consistent state when required.

Consistency requires the programmer to define and implement. The DBMS helps by providing a variety of constraints that it is able to enforce. Requirements beyond these need to be explicitly handled by the programmer.

Isolation is handled by a concurrency control system, that attempts to ensure serialisability while executing transactions concurrently. There are a variety of approaches to achieving this, including various locking schemes and (multi-)versioning schemes. SQL also allows strict isolation to be relaxed when needed.