# T-SQL: Transactions and Concurrency Control

—

September, 2019

1. Transactions
2. Concurrency Controls

Section 1

# Transactions

# Atomicity

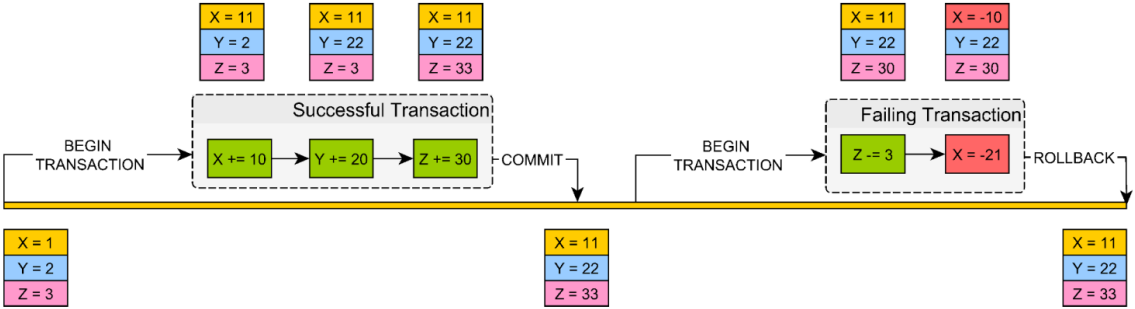


Figure 1: Atomic units of work

# What is a Transaction (Xact)

- Collection of operations that form a single logical unit
- Logical unit:
  - begin transaction ... (SQL) end transaction
- Operations:
  - Read, Write
  - Special actions: begin, commit, rollback

# ACID properties

- Atomicity
  - All action in the Xact happen, or none happen
- Consistency
  - Consistent DB + consistent Xact => consistent DB
- Isolation
  - Execution of one Xact is isolated from that of other Xacts
  - Concurrency control
- Durability
  - If a Xact commits, its effects persist
  - Recovery management

# Autocommit mode in SQL Server (default)

- If not explicitly start a transaction
  - Each statement treated as a seperated tran.
  - If error, automatically rollbacked.
  - Otherwise, automatically committed.

# When to use explicit transactions?

- Two or more action queries affecting related data
- Update foreign key references
- Move rows from one table to another
- Action query whose values inserted depend on previously run select query
- A failure of any set of sql statements would violate integrity

# Example: a transaction to insert invoices and items

```
declare @InvoiceID int;
begin try
        begin tran;
                insert Invoices
                        values (34,'ZXA-080','2016-04-30',14092.59,
                                0,0,3,'2016-05-30',NULL);
                set @InvoiceID = @@IDENTITY;
                insert InvoiceLineItems values (@InvoiceID,1,160,4447.23,
                        'HW upgrade');
                insert InvoiceLineItems values (@InvoiceID,2,167,9645.36,
                        'OS upgrade');
        commit tran;
        print 'transaction committed';
end try
```

# Example: a transaction to insert invoices and items (cont.)

```
begin catch
        print 'error when inserting, rolling back transaction';
        rollback tran;
end catch;
go
```

# T-SQL statement for processing transactions

```
begin {tran | transactions}
save {tran | transactions} 'save_point_name'
commit [tran | transactions]
rollback [[tran | transactions] ['save_point_name']]
```

# Nested transactions

- No true nested transactions
- Use @@trancount counter

`BEGIN TRAN`

- @@trancount += 1

`COMMIT TRAN`

- @@trancount == 1: commit all trans, @@trancount = 0
- @@trancout > 1: nothing committed, @@trancount -= 1

`ROLLBACK TRAN`

- rollback all trans regardless of nesting level
- @@trancount = 0

# Example: nested transactions

```
begin tran;
        print 'First Tran @@trancount: ' + convert(varchar,@@trancount);
        delete Invoices;
        begin tran;
                print 'Second Tran @@trancount:' + convert(varchar,@@trancount
                delete Vendors;
        commit tran;  -- this commit decrements @@trancount
                                -- it doesn't commit 'delete Vendors'
        print 'commit @@trancount: ' + convert(varchar,@@trancount);
rollback tran;
```

# Save points

- Create

```
SAVE TRAN 'save_point_name'
```

- Rollback upto & including the save_point

```
ROLLBACK TRAN 'save_point_name'
```

- Rollback entire transaction

```
ROLLBACK TRAN
```

# Example: a transaction with save_points

```
begin tran
        delete #VendorCopy where VendorID = 1;
        save tran Vendor1;  --1st save_point
                delete #VendorCopy where VendorID = 2;
                save tran Vendor2; --2nd save_point
                        delete #VendorCopy where VendorID = 3;
                        select * from #VendorCopy;
                rollback tran Vendor2;  --rollback 2nd save_point
                select * from #VendorCopy;
        rollback tran Vendor1; --rollback 1st save_point
        select * from #VendorCopy;
commit tran
select * from #VendorCopy;
go
```

Section 2

# Concurrency Controls

# Problems

- Dirty reads
- Lost updates
- Nonrepeatable reads
- Phantom reads

# Dirty reads

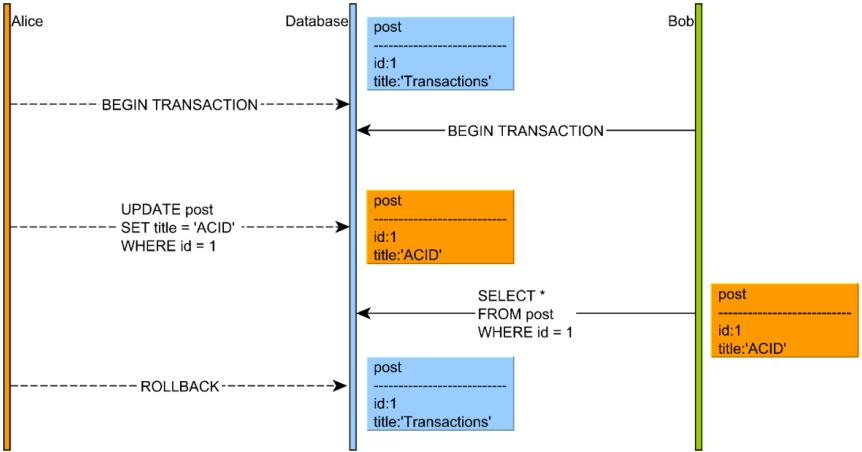


Figure 2: Dirty read

# Lost updates

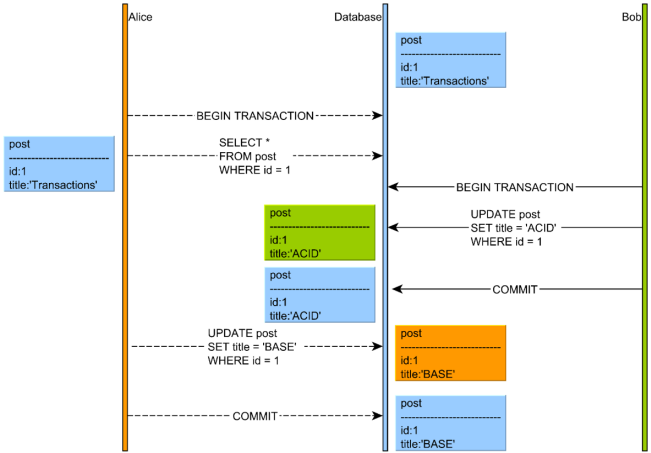


Figure 3: Lost updates

# Nonrepeatable reads

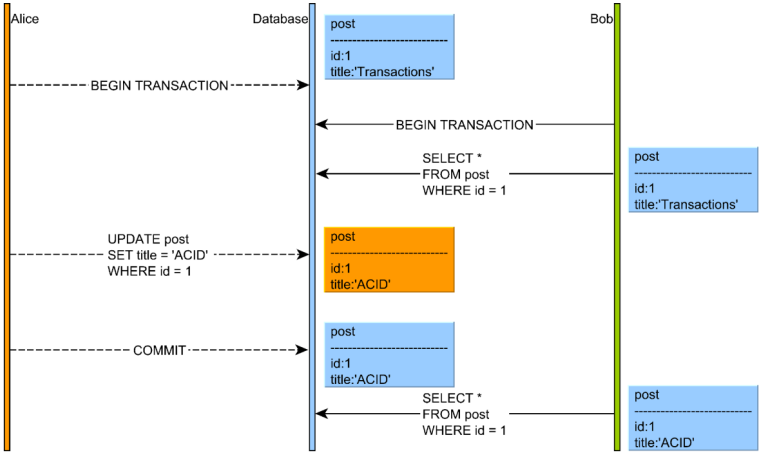


Figure 4: Nonrepeatable reads

# Phantom reads

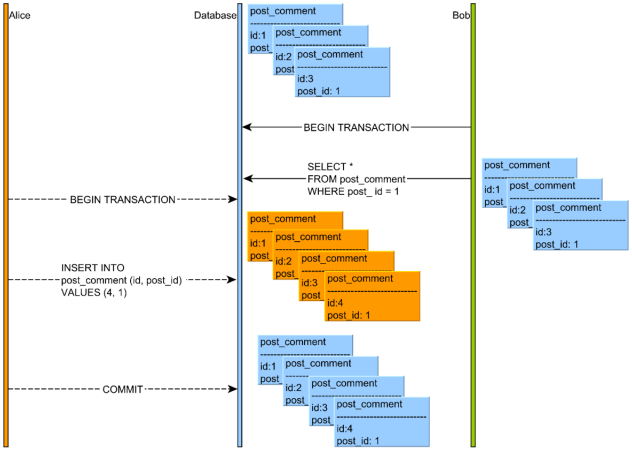


Figure 5: Phantom reads

# Concurrency model for disk-based tables

- Pessimistic concurrency control
  - Locking
  - Default for in-the-box SQL server
- Optimistic concurrency control
  - Row-versioning
    - Linked lists of versions
    - Writers use locks, readers don't
  - Default for Azure

# Pessimistic concurrency control

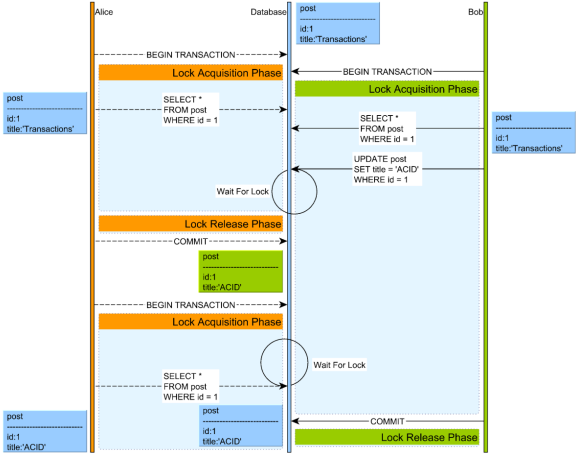


Figure 6: 2PL locking

# Optimistic concurrency control

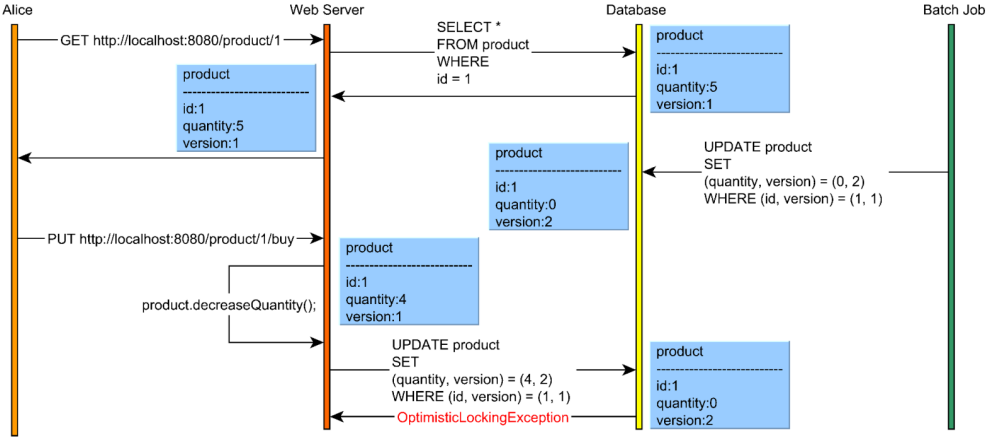


Figure 7: Stateful conversation preventing lost updates

# Isolation Level

| Isolation | Dirty reads | Lost updates | Nonrepeatable reads | Phantoms | Concurrency model | Update conflict detection |
|---|---|---|---|---|---|---|
| Read Uncommitted | Yes | Yes | Yes | Yes | Pessimistic | No |
| Read Committed | No | Yes | Yes | Yes | Pessimistic | No |
| Repeatable Read | No | No | No | Yes | Pessimistic | No |
| Serializable | No | No | No | No | Pessimistic | No |
| Snapshot | No | No | No | No | Optimistic | Yes |
| Read Committed Snapshot | No | Yes | Yes | Yes | Optimistic | No |

Figure 8: Isolation levels

# Isolation level (cont.)

- Default values
  - Recommitted (in-the-box SQL Server)
  - Read committed snapshot (Azure)
- Set isolation level
  - Use table hint
  - Set at session level

    ```
    SET ISOLATION LEVEL 'isolation_level'
    ```
  - Row-versioning flags must be set in SQL Server before use

# Deadlock

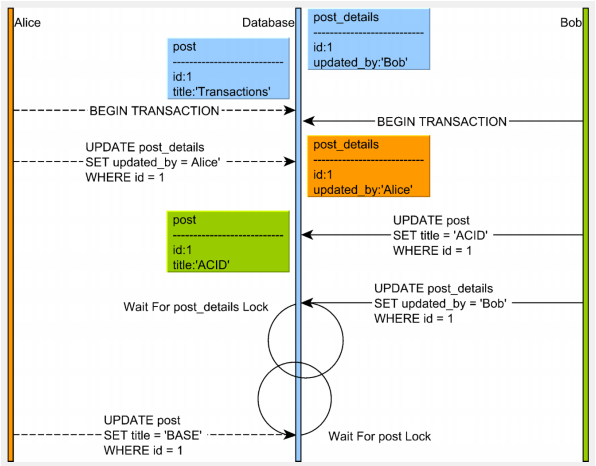


Figure 9: Deadlock

# Deadlock prevention coding techniques

- Don't allow transactions to remain open very long
  - keep transactions short
  - keep select statements outside of the transaction except when absolutery neccessary
  - never code requests for user input during an open transaction
- Use the lowest possible transaction isolation level
  - default level of read committed is almost always sufficient
  - reserve higher levels for short transactions that make changes to data where integrity is vital
- Make large changes when you can be assured of nearly exclusive access
  - not during peak hours
  - manage to get exclusive access when possible

# References


- Bryan Syverson and Murach Joel, SQL Server 2016 for Developers, Mike Murach & Associates Inc., 2016
- Itzik Ben-Gan, Adam Machanic, Dejan Sarka, and Kevin Farlee, T-SQL Querying(Developer Reference), Microsft Press, 2015
- Vlad Mihalcea, High-Performance Java Persistence, Vlad Mihalcea, 2016