

# **NET3000**

# **Database Concepts and SQL**

Instructor: Phil Kaufman

## **Lecture 6**

## **Transactions**

November 2, 2015

# Agenda

- Database integrity
  - ◆ Entity
  - ◆ Domain
  - ◆ Referential
- What is a transaction
- ACID
- Transaction Types
  - ◆ Explicit
  - ◆ Implicit
  - ◆ Automatic

# Agenda (con't)

- Transaction Logging
- Transaction Faults
  - ◆ Lost Update
  - ◆ Dirty Reads
  - ◆ Non-Repeatable Reads
  - ◆ Phantom Reads
- Database Locks
- Transaction Blocking

# Agenda (con't)

## ■ Isolation Levels

- ◆ Read Uncommitted
- ◆ Read Committed
- ◆ Repeatable Read
- ◆ Serializable

## ■ Choosing Isolation Levels

# Database Integrity

## ■ Entity

- ◆ Not allowing multiple rows to have the same identity in a table

## ■ Domain

- ◆ Restricting data to predefined data types (i.e. dates, integers, etc)

## ■ Referential

- ◆ Requires the existence of a related row in another table (i.e. a student for a given student ID must exist)

# What is a Transaction

## ■ A Database Transaction is

- ◆ A unit of work performed within a RDBMS (or similar) against a database and treated in a coherent and reliable way independent of other transactions
- ◆ Provides 2 purposes
  1. provides reliable units of work that **allow correct recovery from failures** and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status
  2. **To provide isolation between programs accessing a database concurrently.** If this isolation is not provided the programs outcome are possibly erroneous

# ACID

## ■ What makes up ACID

### ♦ Atomic

- Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

### ♦ Consistency

- The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

# ACID (con't)

## ■ What makes up ACID (con't)

### ♦ Isolation

- The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control.

### ♦ Durability

- Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter).



# Transaction Types

- **Explicit**
- **Implicit**
- **Automatic**

# Transaction Types (con't)

## ■ Explicit

- ♦ Manually configured by you
- ♦ To begin a transaction:

**BEGIN TRAN[SACTION]**

- ♦ To commit and save a transaction:

**COMMIT [TRAN[SACTION]]**

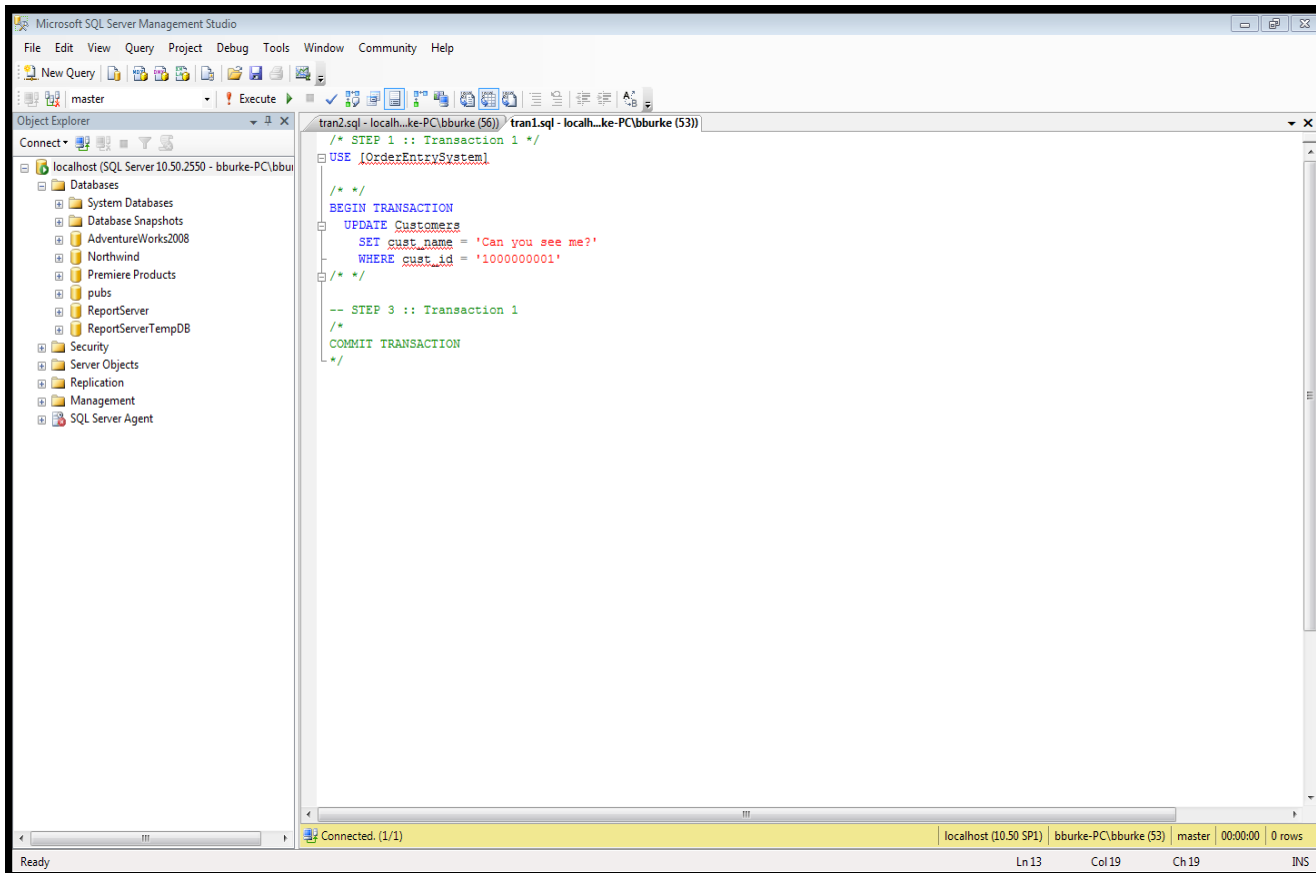
- ♦ To cancel a transaction:

**ROLLBACK [TRAN[SACTION]]**

- ♦ Savepoints
  - ♦ Can be named, saved to and rolled-back to

# Transaction Types (con't)

## ■ Explicit (con't)



# Transaction Types (con't)

## ■ Implicit

- Provided for ANSI compliance
- When enabled, selected T-SQL statements automatically issue a BEGIN TRAN
- You must still explicitly issue a COMMIT or ROLLBACK
- To enable:

**SET IMPLICIT\_TRANSACTIONS ON**

# Transaction Types (con't)

## ■ Automatic

- ♦ Automatically for:
  - ♦ Insert
  - ♦ Update
  - ♦ Delete
- ♦ Each SQL statement is a separate transaction
  - ♦ every Transact-SQL statement is committed (when successful) or rolled back (on error) when it completes
  - ♦ If any one statement fails, it does not affect any other statement (isolation property)
- ♦ \*\* This is the default mode for SQL Server \*\*

# Transaction Logging

- When you modify data, be aware that *every* change is being written to the transaction-log
- This is done for:
  - ◆ Insert
  - ◆ Delete
  - ◆ Update
- Transaction log used in recovery of database

# Insert and Transaction Logging

- A copy of the entire row is written to the transaction-log
- All data is managed in 8 kB storage units called **pages**. The appropriate data pages, containing the records needing to be modified, are located in memory. If these pages are not yet in memory, they are placed in memory from the disk.
- The modifications (insert, update, or delete) are made to the applicable pages in memory.

# INSERT and Transaction Logging (con't)

- The modifications are written to the transaction log.
- The server issues a checkpoint that causes the changed (dirty) pages in memory to be written back to the hard disk. The pages in memory then have their “dirty” flag removed. If the transaction making the changes has been committed, the pages are released and other requests or transactions have access to them. If the checkpoint occurs prior to the transaction being committed, the pages are still locked until the transaction is committed.



# DELETE and Transaction Logging

- A copy of the entire deleted row is written to the transaction-log
- versus the TRUNCATE statement

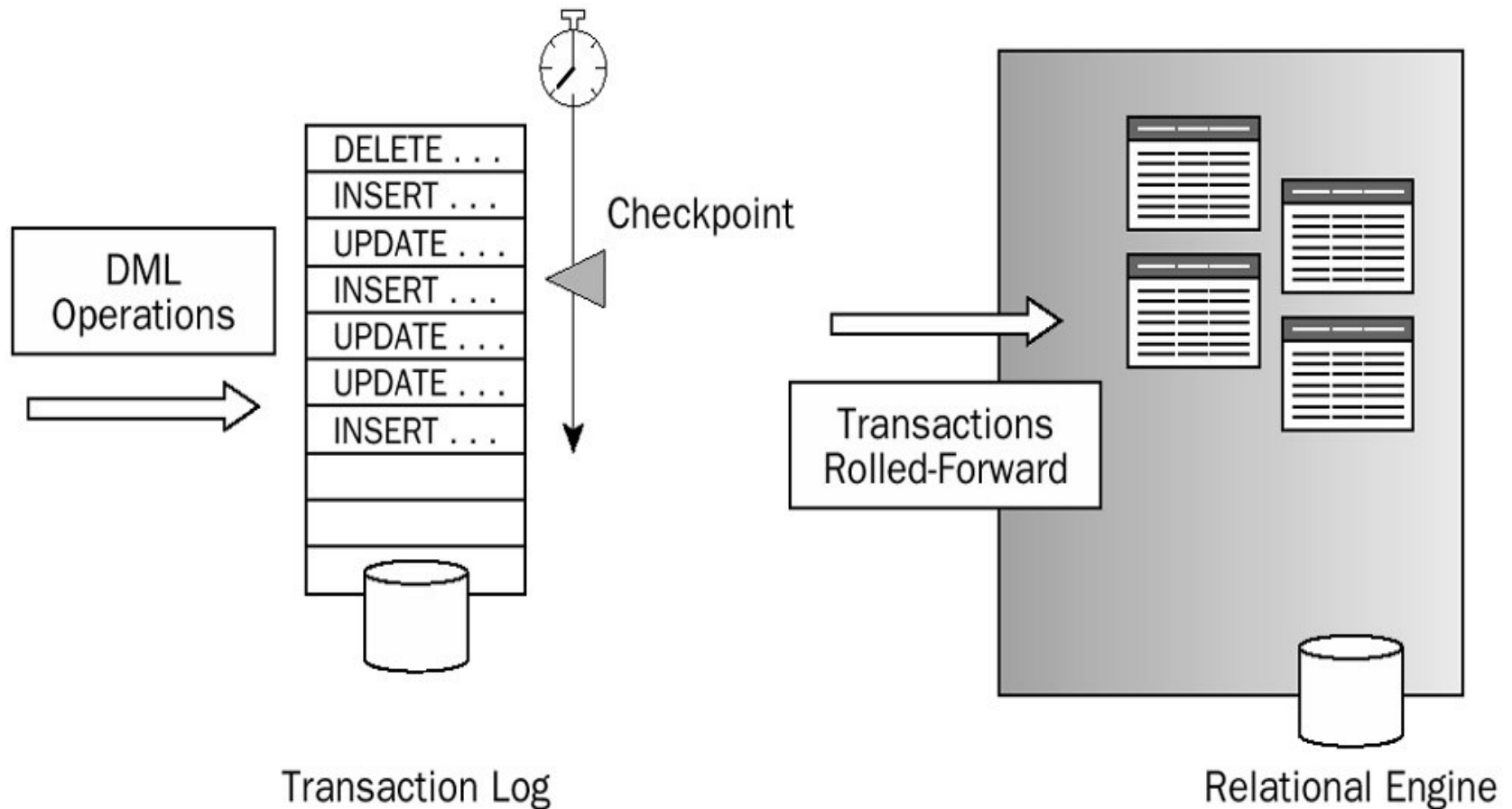
# UPDATE and Transaction Logging

- For some UPDATE operations, only the bytes being changed are logged in transaction logs
- For most UPDATE operations, two log entries are made:
  1. The entire old version of the row is written to the transaction log
  2. The entire new version of the row is written to the transaction log.

# DML commands not Logged

- BCP with FAST option
- SELECT INTO
  - ◆ Think of it as a non-logged INSERT SELECT
- TRUNCATE
  - ◆ Think of it as a non-logged DELETE

# Insert and Transaction Logging (con't)



# Transaction Faults

## ■ Lost Update

- is when one transaction (Transaction #1) reads data into its local memory, and then another transaction (Transaction #2) changes this data and commits its change
- After this, Transaction #1 updates the same data based on what it read into memory before Transaction #2 was executed
- In this case, the update performed by Transaction #2 can be considered a lost update

# Transaction Faults (con't)

## ■ Dirty Read

- if data that has been changed by an open transaction is accessed by another transaction, a dirty read has taken place
- A dirty read can cause problems because it means that a data manipulation language (DML) statement accessed data that logically does not exist yet or will never exist (if the open transaction is rolled back)
- All isolation levels except for read uncommitted protect against dirty reads (we will talk isolation levels momentarily)

# Transaction Faults (con't)

## ■ Non-Repeatable Read

- If a specific set of data is accessed more than once in the same transaction (such as when two different queries against the same table use the same WHERE clause) and the rows accessed between these accesses are updated or deleted by another transaction, a non-repeatable read has taken place
- Therefore, if two queries against the same table with the same WHERE clause are executed in the same transaction, they return different results.
- The repeatable read isolation levels protect a transaction from non-repeatable reads.

# Transaction Faults (con't)

## ■ Phantoms Reads

- Phantom reads are a variation of non-repeatable reads
- A phantom read is when two queries in the same transaction, against the same table, use the same WHERE clause, and the query executed last returns more rows than the first query
- Only the serializable isolation levels protect a transaction from phantom reads



# Database Locks

- Database mechanism for solving concurrency problems:
  - ◆ Database Locks
    - Shared (Read) Locks
    - Exclusive (Write) Locks
    - Update (Read/Write) Locks – select for update
  - ◆ All isolation levels always issue exclusive locks for write operations and hold the locks for the entire duration of the transaction
  - ◆ Typical locking mechanism is **two-phase locking**
  - ◆ Database catalog tables/views provide access to what locks are currently in place

# Transaction Blocking

- Transaction Blocking occurs when
  - when one session holds a lock on a specific resource and a second session attempts to acquire a conflicting lock type on the same resource
  - The second session will block (wait) for first session to release that lock
  - When locking session releases lock, any other session will unblock and continue in their respective transaction grabbing that lock

# Isolation Levels

- Isolation Level determines how transaction integrity is visible between other users
- Isolation Level is set at the database session level
- A program sets this when they ask the database for a connection session
- You can change the isolation level within a session (though typically done at the start of a session)

**SET TRANSACTION ISOLATION LEVEL**

# Isolation Levels (con't)

- Levels are:
  - ◆ Read Uncommitted
  - ◆ Read Committed (SQL Server default)
  - ◆ Repeatable Read
  - ◆ Serializable
- Each of the above levels increase protection against integrity
  - ◆ A lower isolation level increases the ability for all users to access data at the same time, but increases the number of transaction faults (concurrency effects)
  - ◆ A higher isolation level reduces the types of transaction faults but requires more system resources and increases chances of blocking

# Read Uncommitted example

Connection 1	Connection 2
BEGIN TRAN; INSERT Test.TestTran (Col) VALUES (1);	—
—	SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
—	SELECT * FROM Test.TestTran; /* Returns Col = 1 which has not yet been committed, a dirty read. */
UPDATE Test.TestTran SET Col = 2 WHERE Col = 1;	—
—	SELECT * FROM Test.TestTran; /* Returns Col = 2 which has not yet been committed, another dirty read. */
ROLLBACK TRAN;	—
—	SELECT * FROM Test.TestTran; /* Returns an empty result set (which, in this case, is not a dirty read). This shows the problem with the read uncommitted isolation level. The previous 2 statements returned data that logically never existed! */

# Read Committed example

Connection 1	Connection 2
<pre>INSERT Test.TestTran (Col) VALUES (1); BEGIN TRAN;   INSERT Test.TestTran (Col)   VALUES (2);</pre>	<pre>—</pre>
<pre>—</pre>	<pre>BEGIN TRY   SET LOCK_TIMEOUT 0;   SELECT * FROM Test.TestTran; END TRY BEGIN CATCH END CATCH /* Returns only Col = 1. */</pre>
<pre>COMMIT TRAN;</pre>	<pre>—</pre>

# Repeatable Read example

Connection 1	Connection 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN; SELECT * FROM Test.TestTran WHERE Col1 = 1; /* Returns 2 rows. */	—
—	SELECT * FROM Test.TestTran; /* Returns the 2 rows in the table. This state- ment is allowed because it does not change any rows that are included in the transaction in connection 1. */
—	UPDATE Test.TestTran SET Col1 = 2 WHERE Col1 = 1; /* This statement is now blocked by connection 1 because it would cause the transaction in connection 1 to experience a non-repeatable read. */
	SELECT * FROM Test.TestTran WHERE Col1 = 1; Still blocked... /* Still returns 2 rows, i.e. a non-repeatable read did not occur. */
	— /* Abort the above query and execute the following insert statement. */ INSERT Test.TestTran (Col1, Col2) VALUES (1, 3);
	SELECT * FROM Test.TestTran WHERE Col1 = 1; — /* Now returns 3 rows. The rows with the values 1 & 2 for the column Col2 are unchanged. But the row with Col2 = 3 is now included in the result. That row is a phantom read! */
	COMMIT TRAN; —

# Serializable example

Connection 1	Connection 2
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN; SELECT * FROM Test.TestTran WHERE Col1 = 1; /* Returns 2 rows. */	—
—	SELECT * FROM Test.TestTran; /* Returns the 2 rows in the table. This statement is allowed because it does not change any rows that are included in the transaction in connection 1. */
—	UPDATE Test.TestTran SET Col1 = 2 WHERE Col1 = 1; /* This statement is now blocked by connection 1 because it would cause the transaction in connection 1 to experience a non-repeatable read. */
SELECT * FROM Test.TestTran WHERE Col1 = 1; /* Still returns 2 rows, i.e. a non-repeatable read did not occur. */	Still blocked...

—	/* Abort the above query and execute the following insert statement. */ INSERT Test.TestTran (Col1, Col2) VALUES (1, 3); /* This statement is also blocked, exactly as the update was, because it would cause the transaction in connection 1 to experience a phantom read. */
SELECT * FROM Test.TestTran WHERE Col1 = 1; — /* Still returns 2 rows, i.e. a phantom read did not occur. */	—
—	/* Abort the above query and execute the following insert statement. */ INSERT Test.TestTran (Col1, Col2) VALUES (2, 1); /* This statement is also blocked, exactly as the previous insert was. This happened even though the row inserted by this statement would not be included in the query in connection 1. This in turn is because SQL Server found no index to lock a range in, so it locked the entire table. */
COMMIT TRAN;	—
—	The row from the above insert is now completed.



# Choosing Isolation Level

- Use the least-restrictive isolation level possible
  - ◆ Consider whether you can make use of the read uncommitted isolation level
- Avoid use of repeatable read and serializable isolation levels
- Keep transactions as short (in execution duration) as possible
- For exclusive reading of data, use read committed isolation level

# Conclusion

We learned about:

- **Database integrity**
- **What is a transaction**
- **ACID**
- **Transaction Types**
- **Transaction Logging**
- **Transaction Faults**
- **Database Locks**
- **Transaction Blocking**
- **Isolation Levels**
- **Choosing Isolation Levels**