39.1 Transactions

Transactions

A ***transaction*** is a sequence of database operations that must be either completed or rejected as a whole. Partial execution of a transaction results in inconsistent or incorrect data.

Ex: The debit-credit transaction transfers funds from one bank account to another. The first operation removes funds, say $100, from one account, and the second operation deposits $100 in another account. If the first operation succeeds but the second fails, $100 is mysteriously lost. The transaction must complete and save either both operations or neither operation.

Saving complete transaction results in the database is called a ***commit***. Rejecting an incomplete transaction is called a ***rollback***. A rollback reverses the transaction and resets data to initial values. A variety of circumstances cause a rollback:

- The operating system detects a device failure. Ex: Magnetic disk fails during execution of a transaction, and transaction results cannot be written to the database.
- The database detects a conflict between concurrent transactions. Ex: Two airline customers attempt to reserve the same seat on a flight.
- The application program detects an unsuccessful database operation. Ex: In the debit-credit transaction, funds are removed from the debit account, but the credit account is deleted prior to deposit.

When a failure occurs, the database is notified and rolls back the transaction. If the failure is temporary, such as intermittent network problems, the database attempts to restart the transaction. If the failure is persistent, such as a deleted bank account, the database 'kills' the transaction permanently.

PARTICIPATION ACTIVITY 39.1.1: Commit and rollback.

transaction commits

	AccountID	AccountName	BalanceAmount
initial state	A	Maria Rodriguez	11980
	B	Sam Snead	1000
	C	Sam Snead	2900

transaction: subtract $100 from account B
add $100 to account C
commit

transaction rolls back

	AccountID	AccountName	BalanceAmou
	A	Maria Rodriguez	11980
	B	Sam Snead	1000
	C	Sam Snead	2900

subtract $100 from account B
rollback
~~add $100 to account C~~

final state

AccountID	AccountName	BalanceAmount
A	Maria Rodriguez	11980
B	Sam Snead	900
C	Sam Snead	3000

AccountID	AccountName	BalanceAmou
A	Maria Rodriguez	11980
B	Sam Snead	~~900~~ 1000
C	Sam Snead	2900

Animation content:

Static figure:
Two diagrams appear, with captions transaction commits and transaction rolls back. Both diagrams show a transaction between two unnamed tables.

The first diagram, transaction commits, appears as follows:

The upper table has caption initial state. The table has columns AccountID, AccountName, and BalanceAmount, and three rows:
A, Maria Rodriquez, 11980
B, Sam Snead, 1000
C Sam Snead, 2900
Values 1000 and 2900 are highlighted.

The transaction is a rectangle containing three actions:
subtract $100 from account B
add $100 to account C
commit
The commit action is highlighted.

The lower table has caption final state. The table has the same columns as the upper table and three rows:
A, Maria Rodriquez, 11980
B, Sam Snead, 900
C Sam Snead, 3000
Values 900 and 3000 are highlighted.

The second diagram, transaction rolls back, appears as follows:

The upper table is the same as the upper table in the first diagram.

The transaction is a rectangle containing three actions:
subtract $100 from account B
roll back
add $100 to account C
The roll back action is highlighted. The add action is struck out.

The lower table has caption final state. The table has the same columns as the upper table and three rows:
A, Maria Rodriquez, 11980
B, Sam Snead, 1000
C Sam Snead, 2900
Values 1000 and 2900 are highlighted.

Step 1: Initially, Sam Snead has $1,000 in account B and $2,900 in account C. The caption transaction commits appears. The upper table appears

Step 2: The transaction changes balances and commits. Changes are saved in the database. The transaction and lower table appear. The subtract action and the value 900 in the lower table are highlighted. The add action and the value 3000 in the lower table are highlighted.

Step 3: If the operating system, database, or application fails during transaction, the transaction must roll back. The caption transaction rolls back appears. The upper table appears.

Step 4: The database updates account B, detects failure, and restores account B to the initial value $1,000. The transaction and lower table appear. The subtract action and the value 900 in the lower table are highlighted. The rollback action is highlighted and the value 900 is replaced by 1000.

Step 5: After rollback, the transaction terminates. The final operation is not executed, so account C remains $2,900. The add action and the value 2900 in the lower table are highlighted.

Animation captions:

1. Initially, Sam Snead has $1,000 in account B and $2,900 in account C.
2. The transaction changes balances and commits. Changes are saved in the database.
3. If the operating system, database, or application fails during transaction, the transaction must roll back.
4. The database updates account B, detects failure, and restores account B to the initial value $1,000.
5. After rollback, the transaction terminates. The final operation is not executed, so account C

PARTICIPATION ACTIVITY 39.1.2: Transactions.

1) How many SQL statements must be in one transaction?
   - Exactly one
   - At least one
   - At least two

2) After a transaction commits, the transaction can be rolled back:
   - Always
   - Sometimes
   - Never

3) After a rollback, the database restarts a transaction:
   - Always
   - Sometimes
   - Never

ACID properties

All transactions must be atomic, consistent, isolated, and durable, commonly called the ***ACID*** properties:

- In an ***atomic*** transaction, either all or none of the operations are executed and applied to the database. Partial or incomplete results are rolled back, and the database returns to its state prior to execution of the transaction.
- In a ***consistent*** transaction, all rules governing data are valid when the transaction is committed. Completed transactions that violate any rules are rolled back.

  Consistency applies to both universal and business rules. Universal rules apply to all relational data. Ex: Primary keys must be unique and not NULL. Business rules are particular to a specific database or application. Ex: Funds must not be lost in a debit-credit transaction.
- An ***isolated*** transaction is processed without interference from other transactions. Isolated transactions behave as if each transaction were executed one at a time, or serially, when in fact the transactions are processed concurrently.

  Computers usually process multiple transactions concurrently. Multiple processors, or cores, in a single computer might work on multiple transactions in parallel. A single processor might switch to a new transaction while waiting for an active transaction to read or write data.

  Concurrent transactions that access the same data might conflict. Ex: One transaction sums all salaries while another increases all salaries by 10%. If both transactions run concurrently, the

sum might include some increased salaries but not others, and thus the sum might be invalid. To ensure transactions are isolated, databases must prevent conflicts between concurrent transactions.

- A ***durable*** transaction is permanently saved in the database once committed, regardless of system failures.

  System failures potentially cause the loss of transaction data after the transaction is committed. Ex: An application commits a transaction, the transaction data is written to blocks in memory, but hardware fails before the blocks are saved on magnetic disk. Because transaction results are lost, the transaction is not durable.

The ACID properties are supported in two database subsystems. The ***recovery system*** enforces atomic and durable transactions. The ***concurrency system*** enforces isolated transactions. Both the recovery and concurrency systems, along with other database components, support consistency.

Table 39.1.1: Subsystem support for ACID properties.

	Atomic	Consistent	Isolated	Durable
Concurrency system	supporting	supporting	*primary*	none
Recovery system	*primary*	supporting	supporting	*primary*

PARTICIPATION ACTIVITY 39.1.3: ACID properties.

Each example violates an ACID property. Match the property with the violation.

If unable to drag and drop, refresh the page.

Isolated **Durable** **Consistent** **Atomic**

	A transaction increases all employee salaries by 10%. Due to a system failure, increases for only half of the employees are written to the database.

	A transaction saves a row with a foreign key. The foreign key is not NULL and does not match any values of the corresponding primary key.
	Two transactions running in parallel reserve the same seat for different passengers.
	A transaction withdraws $500 from account A and deposits $500 in account B. The withdrawal and deposit are written in the database, but due to a disk drive failure, the information is permanently lost.

Reset

Isolation

Concurrent transactions T_1 and T_2 can conflict in many ways. Common conflicts include dirty read, nonrepeatable read, and phantom read.

In a ***dirty read***, a transaction reads data that has been updated in a second, uncommitted transaction. A dirty read creates invalid results when the second transaction rolls back or makes additional updates to the data. Ex:

1. T_2 updates data X.
2. T_1 reads the updated value of X before T_2 commits.
3. T_2 fails and is rolled back.

Since T_1 reads a value that is eventually rolled back, the result of T_1 is invalid.

In a ***nonrepeatable read***, a transaction repeatedly reads changing data. Ex:

1. T_1 reads data X.
2. T_2 updates X.
3. T_1 rereads X.

If T_1 incorrectly assumes the value of X is stable, the result of T_1 is invalid.

In a ***phantom read***, one transaction inserts or deletes a table row that another transaction is reading. Ex:

1. T_1 begins reading table rows.
2. T_2 inserts a new row into the table.

3. T_1 continues reading table rows.

Since T_1 sees or misses the new row, depending on precisely when T_2 writes the row to the database, the result of T_1 is unpredictable.

To ensure concurrent transactions are isolated, the concurrency system must prevent dirty reads, nonrepeatable reads, phantom reads, and other potential conflicts. Strict prevention of all conflicts, however, increases transaction duration and resource utilization. Most databases can be configured for relaxed enforcement, producing greater efficiency but occasional violations of isolation.

PARTICIPATION ACTIVITY 39.1.4: Dirty, nonrepeatable, and phantom reads.

dirty read

T_1	T_2
upgrade seat	
	read seat class write ticket cost
rollback	
	commit

nonrepeatable read

T_1	T_2
	read seat class write ticket cost
upgrade seat	
	read seat class assign boarding priority
commit	
	commit

phantom read

T_1	T_2
read first class seats	
	reserve first class seat
read economy seats write total seats commit	
	commit

Animation content:

Static figure:
Three examples of concurrent transactions appear, with captions dirty read, nonrepeatable read, and phantom read. The concurrent transactions are named T1 and T2.

The dirty read example shows actions in the following sequence:
T1: upgrade seat
T2: read seat class
T2: write ticket cost
T1: rollback
T2: commit

The nonrepeatable read example shows actions in the following sequence:
T2: read seat class
T2: write ticket cost
T1: upgrade seat
T2: read seat class
T2: assign boarding priority
T1: commit
T2: commit

The phantom read example shows actions in the following sequence:
T1: read first class seats
T2: reserve first class seat
T1: read economy seats
T1: write total seats
T1: commit
T2: commit

Step 1: After T1 upgrades the seat, T2 determines the ticket cost. However, the upgrade is rolled back, so the customer is overcharged. The dirty read example appears.

Step 2: T2 writes the ticket cost based on an economy seat. T1 later upgrades the seat, so the passenger gets first class priority. Cost and priority are inconsistent. The nonrepeatable read example appears.

Step 3: T2 reserves a first class seat after T1 reads the first class seats, so T1 writes an incorrect seat total. The phantom read example appears.

Animation captions:

1. After T_1 upgrades the seat, T_2 determines the ticket cost. However, the upgrade is rolled back, so the customer is overcharged.
2. T_2 writes the ticket cost based on an economy seat. T_1 later upgrades the seat, so the passenger gets first class priority. Cost and priority are inconsistent.
3. T_2 reserves a first class seat after T_1 reads the first class seats, so T_1 writes an incorrect seat total.

PARTICIPATION ACTIVITY 39.1.5: Conflicting transactions.

Match the conflict type with the corresponding transactions.

If unable to drag and drop, refresh the page.

Dirty read | Phantom read | Nonrepeatable read

	T_1 reads salaries of some Accounting department employees T_2 transfers Maria Rodriguez from Accounting to Development T_2 commits T_1 reads salaries of remaining Accounting employees T_1 computes and writes total salary of Accounting employees T_1 commits
	T_2 increases Sam Snead's salary by 20% T_1 reads Sam Snead's salary T_2 rolls back T_1 computes and writes Sam Snead's bonus based on his salary T_1 commits
	T_1 computes total salary for the entire company T_2 increases Sam Snead's salary by 20% T_2 commits T_1 computes total salary by department T_1 writes (department total / company total) for each department T_1 commits

Reset

External resources

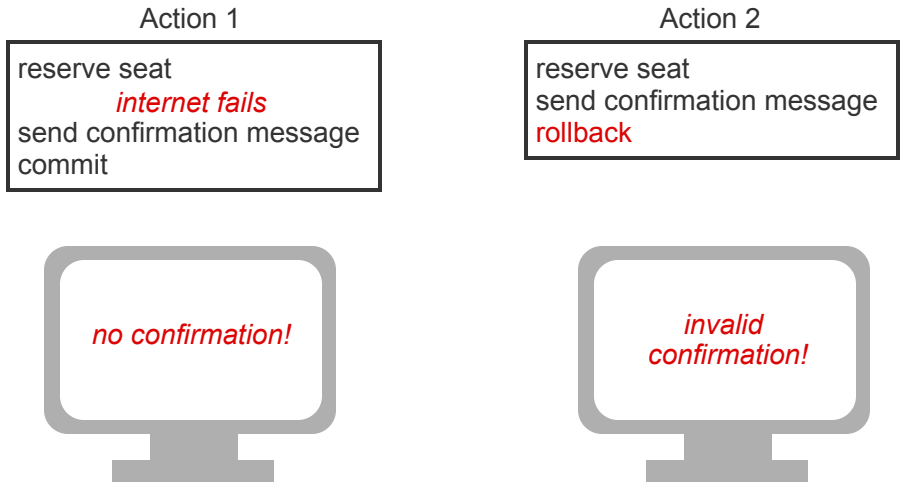
A transaction is a sequence of database operations. A transaction does not affect external resources, such as laptop computers, email systems, or files that are not managed by the database.

An ***action*** is a sequence of software operations that affect the database or external resources. Because application software has limited control over external resources, an action may violate the ACID properties.

Ex: An application executes an action that reserves a seat on a flight. The action commits the reservation in the database and then sends an email confirmation to the customer. If the internet fails, the customer does not receive the confirmation, and the database and email system have different information. The action is not consistent.

Ex: Suppose the action sends the email confirmation to a message server. The message server repeatedly sends the email until the internet is available and the email system confirms receipt. In this scenario, the action is temporarily inconsistent but eventually consistent.

PARTICIPATION ACTIVITY 39.1.6: Actions with external resources.



Animation content:

Static figure:
Two diagrams appear, with captions Action 1 and Action 2. Both diagrams show a transaction and a computer monitor.

The Action 1 diagram shows a transaction with three actions:
Reserve seat
Internet fails
Send confirmation message
Commit
The internet fails action is highlighted.

The computer monitor displays the message "no confirmation".

The Action 2 diagram shows a transaction with three actions:
Reserve seat

Send confirmation message
Rollback
The rollback action is highlighted.

The computer monitor displays the message "invalid confirmation".

Step 1: Action 1 reserves a seat, but the internet fails before a confirmation message is sent. Action 1 appears without the commit action and "no confirmation" message.

Step 2: The database changes commit, but the confirmation is not delivered. The commit action and "no confirmation" message appear.

Step 3: Action 2 reserves a seat, and the confirmation message is delivered. Action 2 appears without the rollback action. The message is "confirmation".

Step 4: The database changes are rolled back. The seat is not reserved, so the confirmation is invalid. The rollback action appears. The message changes to "invalid confirmation".

Animation captions:

1. Action 1 reserves a seat, but the internet fails before a confirmation message is sent.
2. The database changes commit, but the confirmation is not delivered.
3. Action 2 reserves a seat, and the confirmation message is delivered.
4. The database changes are rolled back. The seat is not reserved, so the confirmation is invalid.

Terminology

*In informal communications, **transaction** occasionally means an **action** that affects the database or external resources. In this material, **transaction** always refers to a sequence of database operations.*

PARTICIPATION ACTIVITY 39.1.7: Actions with external resources.

A bank stores checking account data in a database. The bank stores account owner data in a file on a different computer. An action creates a new owner in the file and assigns the owner to a checking account in the database.

1) The action is always atomic.

- True
- False

2) The action is always consistent.

- True
- False

3) The action is always isolated.

- True
- False

4) The action is always durable.

- True
- False

Exploring further:

- Historical perspective on transaction management
- The classic reference book on transaction management, by Jim Gray

CHALLENGE ACTIVITY 39.1.1: Transactions.

539740.3879454.qx3zqy7

Start

Select the ACID property violated in each example.

Pick (a) Transaction 1 slowly updates account A's balance by a complex alg
balance. Transaction 2 quickly withdraws 5 from account A's balanc
before 1 finishes.

Pick (b) A transaction updates a subscription's EndDate. The update is writte

due to a drive failure, the information is permanently lost.

Pick (c) A transaction removes expired subscriptions. Removes for only half are written to the database due to a system failure.

Pick (d) A transaction updates an account with a foreign key. The foreign key values of the corresponding primary key and is not NULL.

1 2

Check Next

39.2 Schedules

Schedules

Often, a database interleaves operations of multiple transactions, in order to utilize computer resources efficiently. A transaction ***schedule*** is a sequential order of operations in multiple transactions.

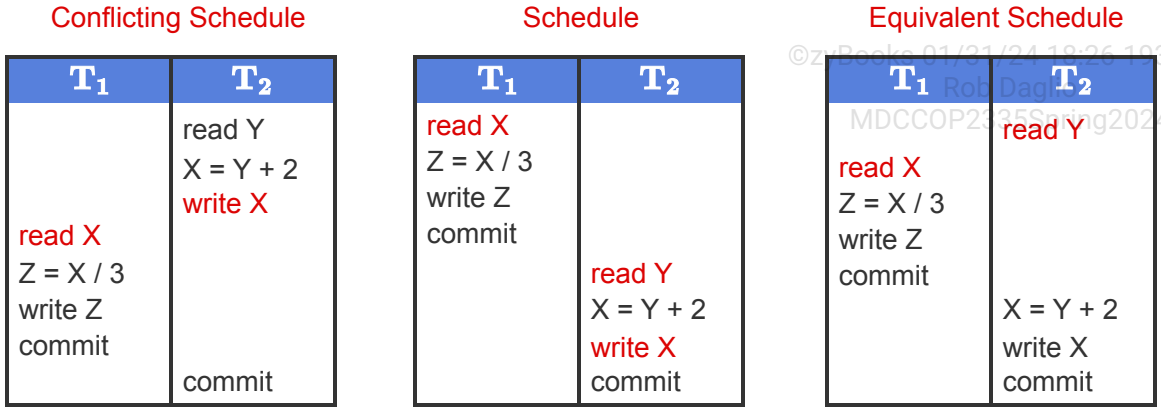
Schedules must preserve the original order of operations within each transaction, as the order within a transaction may affect the result. Operations in different transactions ***conflict*** when the order of the operations may affect the result. Two operations in different transactions conflict when:

- Both operations write the same data.
- One operation reads and another writes the same data.

Two read operations in different transactions never conflict, since the order of reads does not affect the result.

Equivalent schedules contain the same transactions with all conflicting operations in the same order. ***Conflicting schedules*** contain the same transactions with some conflicting operations in different order. Equivalent schedules always have the same result. Conflicting schedules can potentially have different results.

PARTICIPATION ACTIVITY 39.2.1: Equivalent and conflicting schedules.



Animation content:

Static figure:
Three transaction schedules appear, with captions conflicting schedule, schedule, and equivalent schedule. Each schedule shows two concurrent transactions, named T1 and T2. Individual actions of these transactions appear sequentially.

The schedule shows actions in the following sequence:
T1: read X (highlighted)
T1: Z = X / 3
T1: write Z
T1: commit
T2: read Y (highlighted)
T2: X = Y + 2
T2: write X (highlighted)
T2: commit

The conflicting schedule shows actions in the following sequence:
T2: read Y
T2: X = Y + 2
T2: write X (highlighted)
T1: read X (highlighted)
T1: Z = X / 3
T1: write Z
T1: commit
T2: commit

The equivalent schedule shows actions in the following sequence:
T2: read Y (highlighted)
T1: read X (highlighted)
T1: Z = X / 3
T1: write Z
T1: commit
T2: X = Y + 2
T2: write X
T2: commit

Step 1: The schedule has transactions T1 and T2, each with a sequence of operations. The schedule appears without action highlights.

Step 2: Operations read X and write X conflict because the order of the read and write operations affects the final outcome. In schedule, actions T1: read X and T2: read Y are highlighted.

Step 3: In a conflicting schedule, conflicting operations are in a different order. The final value of Z may be different in the conflicting schedule. The conflicting schedule appears with highlighted actions.

Step 4: Operations read X and read Y do not conflict. In schedule, the two read actions are highlighted and the highlight for T2: write X disappears.

Step 5: In an equivalent schedule, non-conflicting operations may be in a different order, but conflicting operations are in the same order. The final value of Z is the same. The equivalent schedule appears with highlights. In schedule, the highlight for T2: write X reappears.

Animation captions:

1. The schedule has transactions T_1 and T_2, each with a sequence of operations.
2. Operations read X and write X conflict because the order of the read and write operations affects the final outcome.
3. In a conflicting schedule, conflicting operations are in a different order. The final value of Z may be different in the conflicting schedule.
4. Operations read X and read Y do not conflict.
5. In an equivalent schedule, non-conflicting operations may be in a different order, but conflicting operations are in the same order. The final value of Z is the same.

PARTICIPATION ACTIVITY 39.2.2: Equivalent and conflicting schedules.

Refer to the schedules in the above animation. Use the initial values below for each question:

X	Y	Z
9	4	0

1) After Schedule executes, what is the value of Z?

Check Show answer

2) After Equivalent Schedule executes, what is the value of Z?

Check Show answer

3) After Conflicting Schedule executes, what is the value of Z?

Check Show answer

4) The _____ Schedule has the same result as Schedule.

Check Show answer

5) The _____ Schedule has a different result than Schedule.

Check Show answer

©zyBooks 01/31/24 18:26 1939727
Rob Daglio
MDCCOP2335Spring2024

Schedules and concurrency

A ***serial schedule*** is a schedule in which transactions are executed one at a time. Serial schedules have no concurrent transactions. Every transaction begins, executes, and commits or rolls back before the next transaction begins. All transactions in a serial schedule are isolated.

Any schedule that is equivalent to a serial schedule is a ***serializable schedule***. A serializable schedule can be transformed into a serial schedule by switching the relative order of reads in different transactions. The order of all operations within a single transaction, and reads and writes of the same data in different transactions, cannot be changed.

Serializable schedules generate the same result as the equivalent serial schedule. Therefore, concurrent transactions in a serializable schedule are isolated.

PARTICIPATION ACTIVITY 39.2.3: Serial and serializable schedules.

Serial Schedule

T_1	T_2
read X	
Z = X / 3	
write Z	
commit	
	read Y
	X = Y + 2
	write X
	commit

Serializable Schedule

T_1	T_2
	read Y
read X	
Z = X / 3	
	X = Y + 2
	write X
	commit
write Z	
commit	

Animation content:

Static figure:
Two transaction schedules appear, with captions serial schedule and serializable schedule. Each schedule shows two concurrent transactions, named T1 and T2. Individual actions of these transactions appear sequentially.

The serial schedule shows actions in the following sequence:
T1: read X (highlighted)
T1: Z = X / 3
T1: write Z
T1: commit
T2: read Y
T2: X = Y + 2
T2: write X
T2: commit

The serializable schedule shows actions in the following sequence:
T2: read Y
T1: read X (highlighted)
T1: Z = X / 3
T2: X = Y + 2
T2: write X (highlighted)
T2: commit
T1: write Z
T1: commit

Step 1: Serial schedules have no concurrent transactions. The T1 transaction commits before T2 starts. The serial schedule appears.

Step 2: The serial schedule is transformed to a new schedule by changing the order of non-conflicting operations. The serial schedule is duplicated. In the duplicate, actions move to the serializable schedule sequence. The first action and last two actions are highlighted.

Step 3: The relative order of all conflicting operations (read X and write X) is unchanged. The new schedule is equivalent to the serial schedule. In the duplicate, the serializable schedule caption appears and highlights revert to the static figure.

Animation captions:

1. Serial schedules have no concurrent transactions. The T_1 transaction commits before T_2 starts.
2. The serial schedule is transformed to a new schedule by changing the order of non-conflicting operations.
3. The relative order of all conflicting operations (read X and write X) is unchanged. The new schedule is equivalent to the serial schedule.

PARTICIPATION ACTIVITY 39.2.4: Serial and serializable schedules.

Match the schedule type to the example schedule.

If unable to drag and drop, refresh the page.

Non-serializable schedule **Serializable schedule** **Serial schedule**

	T_1	T_2
	read X	

Y = X + 4 write Y commit	read X X = X / 8 write X commit

T_1	T_2
read X Y = X + 4	
	read X X = X / 8 write X
write Y commit	
	commit

T_1	T_2
read X X = X + 4	
	read X X = X / 8 write X
write X commit	
	commit

Reset

Isolation levels

Relational databases allow database administrators and application programmers to specify strict or relaxed levels of isolation for each transaction. The SQL standard defines four isolation levels:

1. ***SERIALIZABLE*** transactions run in a serializable schedule with concurrent transactions. Isolation is guaranteed.
2. ***REPEATABLE READ*** transactions read only committed data. After the transaction reads data, other transactions *cannot* update the data. REPEATABLE READ prevents most types of isolation violations but allows phantom reads.
3. ***READ COMMITTED*** transactions read only committed data. After the transaction reads data, other transactions *can* update the data. READ COMMITTED allows nonrepeatable and phantom reads.

4. ***READ UNCOMMITTED*** transactions read uncommitted data. READ UNCOMMITTED processes concurrent transactions efficiently but allows a broad range of isolation violations, including dirty, nonrepeatable, and phantom reads.

Each successive level enables faster processing of concurrent transactions but allows more types of isolation violations. All four levels are supported by most relational databases, but implementation details vary.

Table 39.2.1: Isolation levels.

	Phantom read	Nonrepeatable read	Dirty read
SERIALIZABLE	Not allowed	Not allowed	Not allowed
REPEATABLE READ	Allowed	Not allowed	Not allowed
READ COMMITTED	Allowed	Allowed	Not allowed
READ UNCOMMITTED	Allowed	Allowed	Allowed

PARTICIPATION ACTIVITY 39.2.5: Isolation levels.

1) A SERIALIZABLE transaction can run concurrently with a READ COMMITTED transaction.
   - ○ True
   - ○ False

2) When two READ UNCOMMITTED transactions run concurrently, the result may vary.
   - ○ True
   - ○ False

3) Transactions A and B are both SERIALIZABLE, and A always starts before B. The result may vary.

- ○ True
- ○ False

Schedules and recovery

Serializable schedules affect the concurrency system, which supports isolated transactions. Three additional schedule types affect the recovery system, which supports atomic and durable transactions:

- In a ***nonrecoverable schedule***, one or more transactions cannot be rolled back.
- In a ***cascading schedule***, rollback of one transaction forces rollback of other transactions.
- In a ***strict schedule***, rollback of one transaction *never* forces rollback of other transactions.

In nonrecoverable and cascading schedules, a transaction accesses data written by another uncommitted transaction. In a strict schedule, a transaction *cannot* access data written by uncommitted transactions.

Since rollback is necessary for atomic and durable transactions, nonrecoverable schedules may violate the ACID properties. Therefore, most relational databases detect and prevent nonrecoverable schedules. Cascading schedules do not violate the ACID properties, but cascading rollbacks affect multiple transactions and degrade database performance. Therefore, many databases do not allow cascading schedules.

Most databases allow strict schedules only, which simplifies the recovery system and improves database efficiency.

PARTICIPATION ACTIVITY 39.2.6: Nonrecoverable, cascading, and strict schedules.

Nonrecoverable Schedule

T_1	T_2
write X	
	read X
	commit
~~rollback~~	

Cascading Schedule

T_1	T_2
write X	
	read X
rollback	
	must rollback

Strict Schedule

T_1	T_2
write X	
commit	
	read X
	rollback

Animation content:

Static figure:
Three transaction schedules appear, with captions nonrecoverable schedule, cascading schedule, and strict schedule. Each schedule shows two concurrent transactions, named T1 and T2. Individual actions of these transactions appear sequentially.

The nonrecoverable schedule shows actions in the following sequence:
T1: write X
T2: read X
T2: commit
T1: rollback (highlighted and struck out)

The cascading schedule shows actions in the following sequence:
T1: write X
T2: read X
T2: rollback
T1: must rollback (highlighted and italicized)

The strict schedule shows actions in the following sequence:
T1: write X
T1: commit
T2: read X
T2: rollback (highlighted)

Step 1: Rolling back T1 would change data that T2 has read. Since T2 has committed, T1 cannot roll back. The nonrecoverable schedule appears.

Step 2: Rolling back T1 changes data read by T2. T2 must also rollback. The cascading schedule appears.

Step 3: T2 reads after T1 commits. Rollback of T2 does not cascade to T1. The strict schedule appears.

Animation captions:

1. Rolling back T_1 would change data that T_2 has read. Since T_2 has committed, T_1 cannot roll back.
2. Rolling back T_1 changes data read by T_2. T_2 must also rollback.
3. T_2 reads after T_1 commits. Rollback of T_2 does not cascade to T_1.

PARTICIPATION ACTIVITY 39.2.7: Schedules and recovery.

Match the schedule type to the description.

If unable to drag and drop, refresh the page.

Cascading **Strict** **Nonrecoverable**

	Transaction A writes data and rolls back before transaction B reads the data.
	Transaction A writes data. Transaction B reads the data and commits before transaction A commits.
	Transaction A writes data. Transaction B reads the data. Transaction A rolls back before B commits.

Reset

CHALLENGE ACTIVITY 39.2.1: Schedules.

539740.3879454.qx3zqy7

Start

Enter the value of Z after each schedule executes. Initial values: X = 4, Y = 2, Z = 0.

Schedule A

T_1	T_2
read X	
Z = X * 2	
write Z	
commit	
	read Y
	X = Y + 4

Schedule B

T_1	T_2
read Y	
X = Y + 4	
write X	
	read X
	Z = X * 2
	write Z

Schedule C

T_1	T_2
	read Y
read X	
Z = X * 2	
write Z	
commit	X = Y + 4

	write X
	commit

Z = Ex: 5

	commit
commit	

Z =

	write X
	commit

Z =

A and B are Pick schedules.

A and C are Pick schedules.

B and C are Pick schedules.

1

Check Try again

39.3 Concurrency

Locking

The ***concurrency system*** is a database component that manages concurrent transactions. The concurrency system implements isolation levels while attempting to optimize overall database performance. Tradeoffs between isolation levels and performance are complex, and many techniques have been implemented in concurrency systems. The most common technique is locking.

A ***lock*** is permission for one transaction to read or write data. Concurrent transactions are prevented from reading or writing the same data. A transaction takes locks when the transaction needs to read or write data. A transaction releases locks when the transaction is committed or no longer needs the locked data.

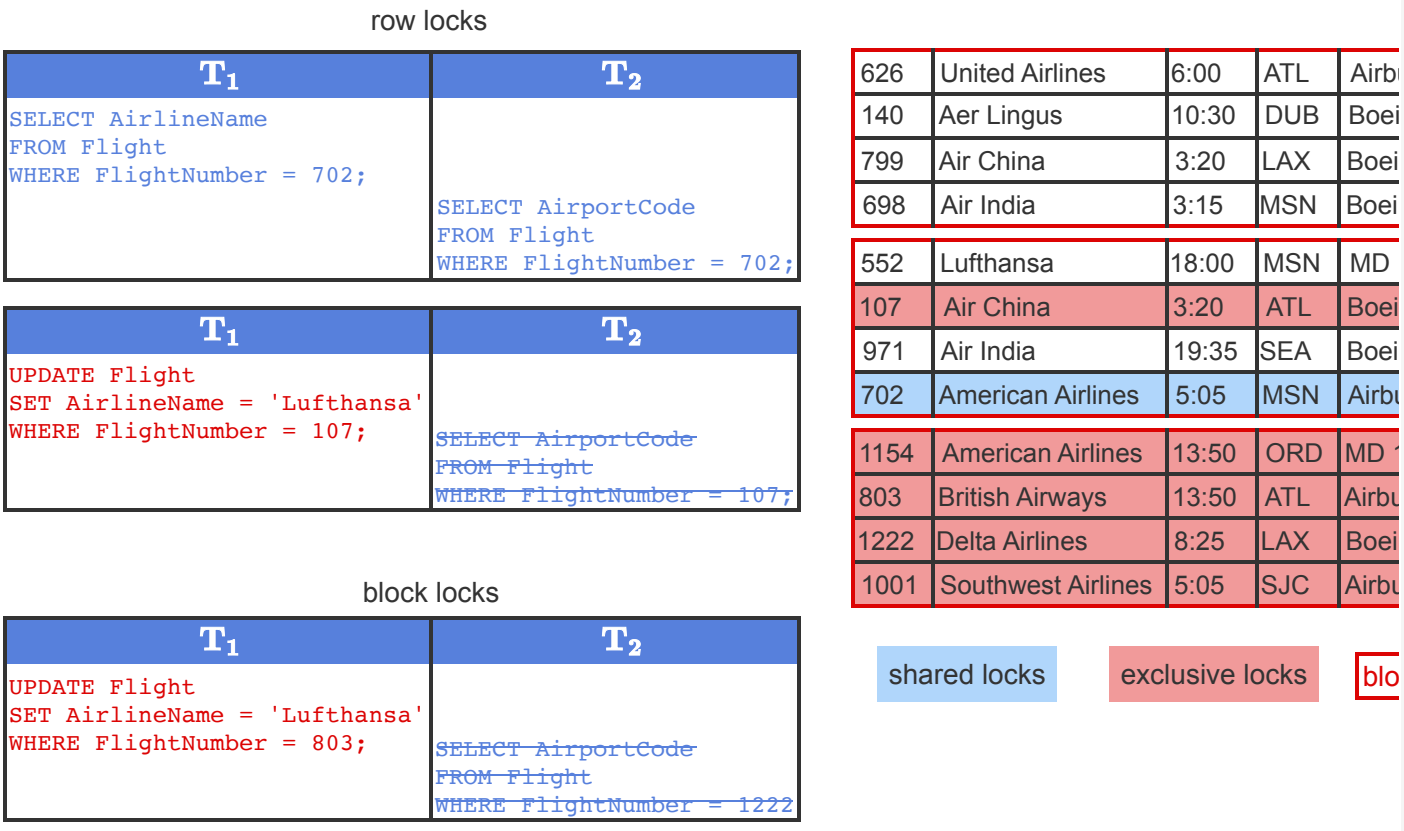
A ***shared lock*** allows a transaction to read, but not write, data. An ***exclusive lock*** allows a transaction to read and write data. Concurrent transactions can hold shared locks on the same data. When one transaction holds an exclusive lock, however, no concurrent transaction can take a shared or exclusive lock on the same data.

Lock scope is the collection of data reserved by a lock. Lock scope is often a single row, allowing other transactions to access other rows in the same table. If a transaction needs access to multiple rows, lock scope might be a block or the entire table. Since transactions also read and write indexes, lock scope might be an index entry, index block, or entire index.

The concurrency system monitors active transactions, determines when locks are needed, and issues requests to grant and release locks. Requests are sent to the ***lock manager***, a component of the concurrency system that tracks, grants, and releases locks.

By requesting shared and exclusive locks as needed, the concurrency system implements the isolation level specified for each transaction. By minimizing lock scope and duration, the concurrency system reduces the duration of transactions that are waiting for locked data.

PARTICIPATION ACTIVITY 39.3.1: Shared and exclusive locking.



Animation content:

Static figure:
Three transaction schedules appear.

The first schedule has caption row lock and actions in the following sequence:
T1: SELECT AirlineName FROM Flight WHERE FlightNumber = 702;
T2: SELECT AirportCode FROM Flight WHERE FlightNumber = 702;

The second schedule has caption row lock and actions in the following sequence:
T1: UPDATE Flight SET AirlineName = 'Lufthansa' WHERE FlightNumber = 107;
T2: SELECT AirportCode FROM Flight WHERE FlightNumber = 107;
The last action in this schedule is struck out.

The third schedule has caption block lock and actions in the following sequence:
T1: UPDATE Flight SET AirlineName = 'Lufthansa' WHERE FlightNumber = 803;
T2: SELECT AirportCode FROM Flight WHERE FlightNumber = 1222;
The last action in this schedule is struck out.

Three blocks of data appear, with columns flight number, airline name, departure time, airport code, and aircraft type. Each block has four rows of data. Rows are sorted on airport code. Highlights indicate a shared lock on one row and exclusive locks on another row and one block.

Step 1: The concurrent transactions T1 and T2 can take a shared lock on the same row. The shared lock allows the transactions to read the row with FlightNumber 702. The first schedule appears. The fourth row of the second block takes a shared lock:
702, American Airlines, 5:05, MSN, Airbus 323

Step 2: If T1 takes an exclusive lock on the row with FlightNumber 107, T2 cannot access the row until the lock is released. The second schedule appears. The second row of the second block takes an exclusive lock:
107, Air China, 3:30, AT, Boeing 787
The T2 SELECT action, which attempts to read flight number 107, is struck out.

Step 3: If T1 takes an exclusive lock on a block, T2 cannot access any row in the block until the lock is released. The third schedule appears. The third block contains the following rows:
803, British Airways, 13:50, ATL, Airbus 323
1222, Delta Airlines, 8:25, LAX, Boeing 747
The third block takes an exclusive lock. The T2 SELECT action, which attempts to read fight number 1222, is struck out.

Animation captions:

1. The concurrent transactions T_1 and T_2 can take a shared lock on the same row. The shared lock allows the transactions to read the row with FlightNumber 702.
2. If T_1 takes an exclusive lock on the row with FlightNumber 107, T_2 cannot access the row until the lock is released.
3. If T_1 takes an exclusive lock on a block, T_2 cannot access any row in the block until the lock is released.

PARTICIPATION ACTIVITY 39.3.2: Locking.

1) Several transactions can hold concurrent shared locks on the same row.
   - True
   - False

2) Several transactions can hold concurrent exclusive locks on the same row.
   - True
   - False

3) One transaction can hold an exclusive lock while other transactions hold shared locks on the same row.
   - True
   - False

4) Several transactions can hold concurrent exclusive locks on the same block, as long as the transactions access different rows in the block.
   - True
   - False

5) When a transaction takes a shared lock on a block, other transactions may be delayed.
   - True
   - False

Two-phase locking

When a transaction's isolation level is set to SERIALIZABLE, a transaction is fully isolated from other transactions. ***Two-phase locking*** is a specific locking technique that ensures serializable transactions. Three variations of two-phase locking are common in relational databases:

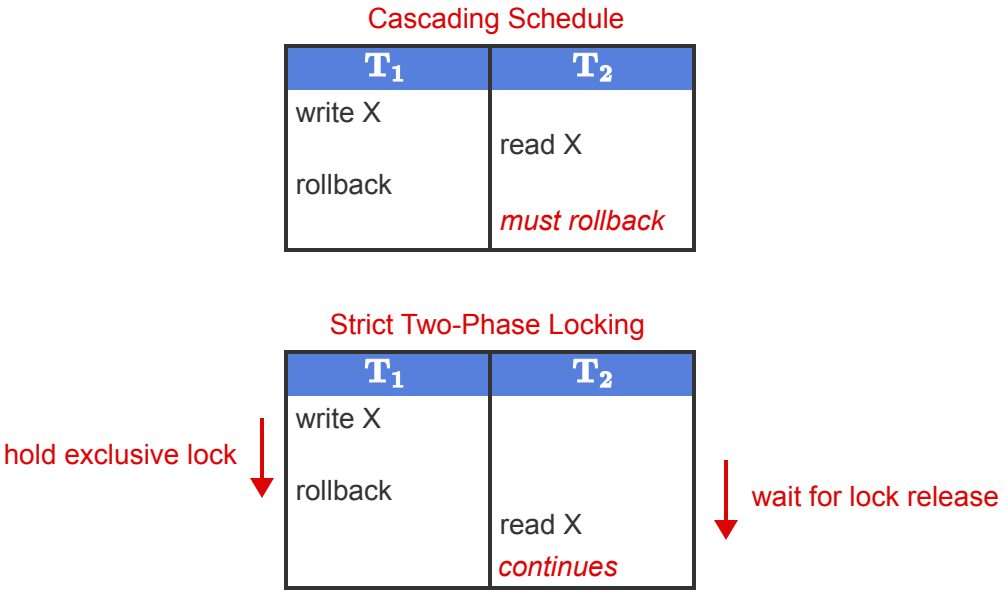
- ***Basic two-phase locking*** has expand and contract phases for each transaction, also known as grow and shrink phases. In the expand phase, the transaction can take, but not release, locks. In the contract phase, the transaction can release, but not take, locks.

- ***Strict two-phase locking*** holds all exclusive locks until the transaction commits or rolls back. The expand phase is the same as in basic two-phase locking, but the contract phase releases only shared locks.
- ***Rigorous two-phase locking*** holds both shared and exclusive locks until the transaction commits or rolls back. In effect, rigorous two-phase locking has no contract phase.

All three variations prevent conflicts and ensure serializable transactions. Strict and rigorous also prevent cascading rollbacks while basic does not. Rigorous is easier to implement than strict, but less efficient, since rigorous holds shared locks longer.

Concurrency systems in most relational databases implement strict two-phase locking.

PARTICIPATION ACTIVITY 39.3.3: Strict two-phase locking.



Animation content:

Static figure:
Two transaction schedules appear.

The first schedule has caption cascading schedule and actions in the following sequence:
T1: write X
T2: read X
T1: rollback
T2: must rollback (highlighted and italicized)

The second schedule has caption strict two-phase locking and actions in the following sequence:

T1: write X
T1: rollback
T2: read X
T2: continues (highlighted and italicized)
An arrow, with caption hold exclusive lock, extends from the write X action to the rollback action. An arrow, with caption wait for lock release, extends from the rollback action to the continues action.

Step 1: Rollback of T1 forces rollback of T2, so the schedule is cascading. The cascading schedule appears.

Step 2: With strict two-phase locking, exclusive locks are not released during the contract phase. The exclusive lock is held until T1 rolls back. The strict two-phase locking schedule appears, with actions in the same sequence as the cascading schedule. An arrow, with caption hold exclusive lock, appears. The arrow extends from the write X action to the rollback action.

Step 3: T2 read must wait until the exclusive lock is released. The strict two-phase locking actions change to:
T1: write X
T1: rollback
T2: read X
An arrow, with caption wait for lock release, appears. The arrow extends from the rollback action to the read X action.

Step 4: Rollback of T1 does not cascade to T2. The T2: continues action is added to the end of the strict two-phase locking schedule.

Animation captions:

1. Rollback of T_1 forces rollback of T_2, so the schedule is cascading.
2. With strict two-phase locking, exclusive locks are not released during the contract phase. The exclusive lock is held until T_1 rolls back.
3. T_2 read must wait until the exclusive lock is released.
4. Rollback of T_1 does not cascade to T_2.

PARTICIPATION ACTIVITY 39.3.4: Two-phase locking.

1) Which two-phase locking technique results in the longest wait times for concurrent transactions?
   - Basic
   - Strict
   - Rigorous

2) Which two-phase locking technique has, in effect, just one phase?
   - Basic
   - Strict
   - Rigorous

3) A database uses *basic* two-phase locking. Transaction A takes an exclusive lock on X. Transaction B requests a shared lock on X. When is the shared lock granted?
   - During the contract phase of A
   - Immediately, when B requests the shared lock
   - After A commits or rolls back

4) A database uses *strict* two-phase locking. Transaction A takes an exclusive lock on X. Transaction B requests a shared lock on X. When is the shared lock granted?
   - During the contract phase of A
   - Immediately, when B requests the shared lock
   - After A commits or rolls back

Deadlock

Deadlock is a state in which a group of transactions are frozen. In a deadlock, all transactions request access to data that is locked by another transaction in the group. None of the transactions can proceed. Ex:

1. T_1 takes an exclusive lock on X.

2. T_2 takes an exclusive lock on Y.
3. T_1 requests a shared lock on Y.
4. T_2 requests a shared lock on X.

T_1 waits for T_2 to release the lock on Y. T_2 waits for T_1 to release the lock on X. The transactions are deadlocked.

A ***dependent transaction*** is waiting for data locked by another transaction. A ***cycle*** of dependent transactions indicates deadlock has occurred. Ex: T_1 depends on T_2, which depends on T_3, which depends on T_1. The transactions are deadlocked.

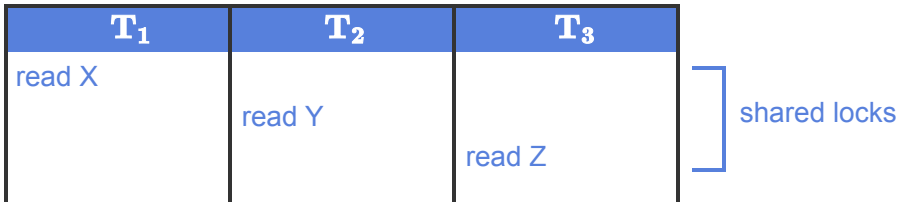
Concurrency systems manage deadlock with a variety of techniques:

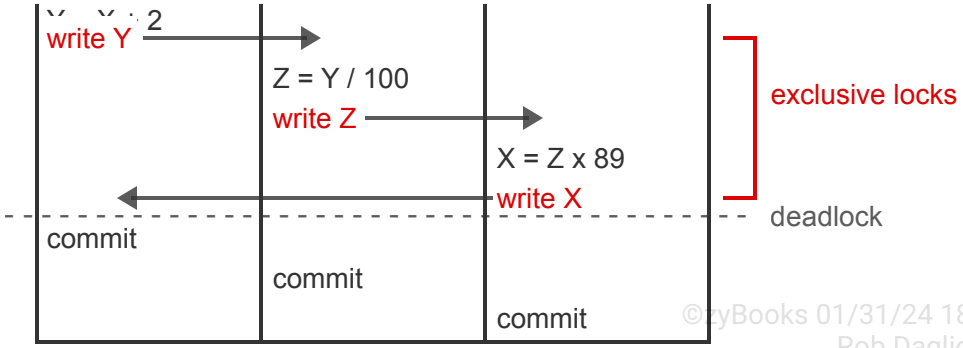
- ***Aggressive locking***. Each transaction requests all locks when the transaction starts. If all locks are granted, the transaction runs to completion. If not, the transaction waits until other transactions release locks. Either way, the transaction cannot participate in a deadlock.
- ***Data ordering***. All data needed by concurrent transactions is ordered, and each transaction takes locks in order. Taking locks in order prevents deadlock. Ex: The sequence above is modified so that locks on X precede locks on Y:

  1. T_1 takes an exclusive lock on X.
  2. T_2 requests a shared lock on X.
  3. T_1 takes a shared lock on Y.
  4. T_2 takes an exclusive lock on Y.

  Request 2 waits for T_1 to release the exclusive lock on X. T_1 proceeds and eventually releases the lock on X. Now request 2 is granted and T_2 proceeds.
- ***Timeout***. When waiting time for a lock exceeds a fixed period, the transaction requesting the lock rolls back. Alternatively, the concurrency system compares transaction start times and rolls back the later transaction. The timeout period is set by the database or configured by the database administrator.
- ***Cycle detection***. The concurrency system periodically checks for cycles of dependent transactions. When a cycle is detected, the concurrency system selects and rolls back the 'cheapest' transaction. The cheapest transaction might, for example, have the fewest rows locked or most recent start time. The rollback breaks the deadlock.

PARTICIPATION ACTIVITY 39.3.5: Cycle of dependent transactions.





Animation content:

Static figure:
A schedule has three transactions and actions in the following sequence:
T1: read X
T2: read Y
T3: read Z
T1: Y = X + 2
T1: write Y
T2: Z = Y / 100
T2: write Z
T3: X = Z x 89
T3: write X
T1: commit
T2: commit
T3: commit

The three read actions have caption shared locks. The three write actions have caption exclusive locks. A dashed line above the commit actions has caption deadlock. Arrows indicate each exclusive lock blocks another transaction:
T1: write Y blocks T2
T2: write Z blocks T3
T3: write X blocks T1

Step 1: The schedule has three transactions that read and write X, Y, and Z. The schedule appears.

Step 2: Transactions first take shared locks when reading, then request exclusive locks when writing. The read actions are labeled shared locks. The write actions are labeled exclusive locks.

Step 3: Since T2 has a shared lock on Y, the T1 exclusive lock request for Y waits for T2 to commit and release the shared lock. An arrow appears, indicating T1: write Y blocks T2.

Step 4: Similarly, T2 waits for T3 to release the shared lock on Z, and T3 waits for T1 to release the

shared lock on X. Two more arrows appear, indicating T2: write Z blocks T3 and T3: write X blocks T1.

Step 5: The dependency cycle causes deadlock. A dashed line appears above the commit actions, with caption deadlock.

Animation captions:

1. The schedule has three transactions that read and write X, Y, and Z.
2. Transactions first take shared locks when reading, then request exclusive locks when writing.
3. Since T_2 has a shared lock on Y, the T_1 exclusive lock request for Y waits for T_2 to commit and release the shared lock.
4. Similarly, T_2 waits for T_3 to release the shared lock on Z, and T_3 waits for T_1 to release the shared lock on X.
5. The dependency cycle causes deadlock.

PARTICIPATION ACTIVITY 39.3.6: Deadlock.

1) Refer to the above animation. How many transactions must roll back to break the deadlock?
   - One
   - Two
   - Three

2) Whenever deadlock occurs, a cycle of dependent transactions always exists.
   - True
   - False

3) Deadlock can occur in a serializable schedule.
   - True
   - False

4) A transaction with isolation level SERIALIZABLE can participate in a deadlock.
   - True
   - False

5) Which deadlock management technique delays the fewest possible transactions?

- Aggressive locking
- Data ordering
- Timeout
- Cycle detection

Snapshot isolation

Exclusive locks prevent concurrent transactions from accessing data and therefore increase transaction duration. For this reason, many databases support alternative concurrency techniques. ***Optimistic techniques*** execute concurrent transactions without locks and, instead, detect and resolve conflicts when transactions commit. Optimistic techniques are effective when conflicts are infrequent, as in analytic applications with many reads and few updates.

One leading optimistic technique is snapshot isolation. ***Snapshot isolation*** creates a private copy of all data accessed by a transaction, called a ***snapshot***, as follows:

1. Create a snapshot when the transaction starts.
2. Apply updates to the snapshot rather than the database.
3. Prior to commit, check for conflicts. A conflict occurs when concurrent transactions write the same data. Reads do not cause conflicts.
4. If no conflict is detected, write snapshot updates to the database (commit).
5. If a conflict is detected, discard the snapshot and restart the transaction (roll back).

Since snapshot isolation does not take locks, transactions never wait. However, transactions occasionally restart after all operations are processed.

Two-phase locking is the most common concurrency technique, implemented in most relational databases. Snapshot isolation is a widely used alternative, available in Oracle and SQL Server.

PARTICIPATION ACTIVITY 39.3.7: Snapshot isolation vs. two-phase locking.

Snapshot Isolation

T_1	T_2
read X	
	read Y
Y = X + 2	
	Z = Y / 100
write Y	

take snapshots

Two-Phase Locking

T_1	T_2
read X	
	read Y
Y = X + 2	
	Z = Y / 100
write Y →	

Animation content:

Static figure:
Two schedules appear, with captions snapshot isolation and two-phase locking. Both schedules have the same sequence of actions:
T1: read X
T2: read Y
T1: Y = X + 2
T2: Z = Y / 100
T1: write X
T2: write Z
T1: commit
T2: commit

The snapshot isolation schedule has caption neither transaction waits. The read actions have caption take snapshots. The commit actions have caption no conflict detected.

The two-phase locking schedule has caption T1 waits for T2. An arrow indicates T1: write X blocks T2.

Step 1: Under snapshot isolation, the transactions take snapshots and process in parallel. The snapshot isolation schedule appears. The take snapshots caption appears.

Step 2: Since the transactions write different data, no conflict is detected. Both transactions comm The write actions are highlighted. The no conflict detected caption appears.

Step 3: Under snapshot isolation, neither transaction waits. The caption neither transaction waits appears.

Step 4: Under two-phase locking, T1 waits for T2 to commit and release the shared lock on Y. The two-phase locking schedule, caption, and arrow appear.

Animation captions:

1. Under snapshot isolation, the transactions take snapshots and process in parallel.
2. Since the transactions write different data, no conflict is detected. Both transactions commit.
3. Under snapshot isolation, neither transaction waits.
4. Under two-phase locking, T_1 waits for T_2 to commit and release the shared lock on Y.

When isolation level is set to SERIALIZABLE, two-phase locking ensures serializable schedules. Snapshot isolation does not ensure serializable schedules and may generate unexpected results. As a result, some databases implement a modified version of snapshot isolation.

Serializable snapshot isolation extends standard snapshot isolation and ensures serializable schedules when isolation level is set to SERIALIZABLE. Postgres supports serializable snapshot isolation.

PARTICIPATION ACTIVITY 39.3.8: Snapshot isolation allows non-serializable schedule.

Serial Schedule

T_1	T_2
read X	
Y = X	
write Y	
commit	
	read Y
	X = Y
	write X
	commit

transactions commit
X, Y have same value

Non-Serializable Schedule

T_1	T_2
read X	
	read Y
Y = X	
	X = Y
write Y	
	write X
commit	
	commit

Two-Phase Locking
deadlock

Snapshot Isolation
transactions commit
X, Y swap values

Animation content:

Static figure:
Two schedules appear, with captions serial schedule and non-serializable schedule.

The serial schedule has actions in the following sequence:
T1: read X
T1: Y = X
T1: write Y
T1: commit
T2: read Y
T2: X = Y
T2: write X
T2: commit

The serial schedule has one caption:
Transactions commit: X, Y have same value

The non-serializable schedule has actions in the following sequence:
T1: read X
T2: read Y
T1: Y = X
T2: X = Y
T1: write Y
T2: write X
T1: commit
T2: commit

The non-serializable schedule has two captions:
Two-phase locking: deadlock
Snapshot isolation: transactions commit / X, Y swap values

Step 1: The serial schedule commits and results in the same value for X and Y. The serial schedule appears, with caption Transactions commit.

Step 2: The non-serializable schedule contains conflicting operations in a different order than the serializable schedule. The non-serializable schedule appears. Actions T1: write Y and T2: read Y are highlighted, indicating the order has reversed.

Step 3: The non-serializable schedule contains a cycle and results in deadlock under two-phase locking. In the non-serializable schedule, the Two-phase locking caption appears. An arrow indicates T1: write Y blocks T2. Another arrow indicates T2: write X blocks T1.

Step 4: Under snapshot isolation, no conflict is detected, and the transactions commit. In the non-serializable schedule, the Snapshot isolation caption appears. The arrows disappear.

Step 5: Snapshot isolation results in different values for X and Y. The caption X, Y swap values appears.

Animation captions:

1. The serial schedule commits and results in the same value for X and Y.
2. The non-serializable schedule contains conflicting operations in a different order than the serializable schedule.
3. The non-serializable schedule contains a cycle and results in deadlock under two-phase locking.
4. Under snapshot isolation, no conflict is detected, and the transactions commit.
5. Snapshot isolation results in different values for X and Y.

PARTICIPATION ACTIVITY 39.3.9: Snapshot isolation.

Place snapshot isolation steps in the correct order.

If unable to drag and drop, refresh the page.

- Determine if any updates conflict with other transactions
- Write updates to a private copy of data
- Write updates to the database or roll back the transaction
- Make a private copy of data accessed by the transaction

	Step 1
	Step 2
	Step 3
	Step 4

Reset

CHALLENGE ACTIVITY 39.3.1: Concurrency.

539740.3879454.qx3zqy7

Start

T_1	T_2

`SELECT CountryName`
`FROM Country`
`WHERE CountryID = 434;`
	`UPDATE Country`
	`SET Area = 88900.38464`
	`WHERE CountryID = 642;`

If lock scope is a single row, and T_1 takes an exclusive lock on CountryID 434, what happens

Pick

If lock scope is a block, and T_1 takes an exclusive lock on CountryID 434, select the row(s) th

☐	642	Romania	Bucharest	88834.38464	Europe
☐	854	Burkina Faso	Ouagadougou	105637.5506	Africa
☐	434	Libya	Tripoli	679362.192	Africa
☐	360	Indonesia	Jakarta	699451.0874	Asia
☐	422	Lebanon	Beirut	3949.825082	Asia
☐	578	Norway	Oslo	140974.7814	Europe
☐	208	Denmark	Copenhagen	16212.42964	Europe
☐	548	Vanuatu	Port Vila	4706.585313	Oceania
☐	232	Eritrea	Asmara	38996.31801	Africa

1 2 3 4

Check Next

39.4 Recovery

Failure scenarios

The recovery system supports atomic transactions by ensuring partial transaction results are not saved in a database. The recovery system supports durable transactions by ensuring committed transactions are not lost due to hardware or software failures.

The recovery system must manage three failure scenarios:

1. A ***transaction failure*** results in a rollback. The application program may initiate a rollback due to logical errors. The database system may initiate rollback due to deadlock or insufficient disk space. The operating system may initiate rollback if a hardware or software component fails. Regardless of the cause, the recovery system restores all data changed by the transaction to the original values.
2. A ***system failure*** includes a variety of events resulting in the loss of main memory. Databases initially write to main memory blocks, which are lost when an application, the operating system, or the database system fails. Blocks are subsequently saved on storage media, which normally

survive a system failure. If main memory is lost before blocks are written to storage media, data written by committed transactions might be lost. In this event, the recovery system:

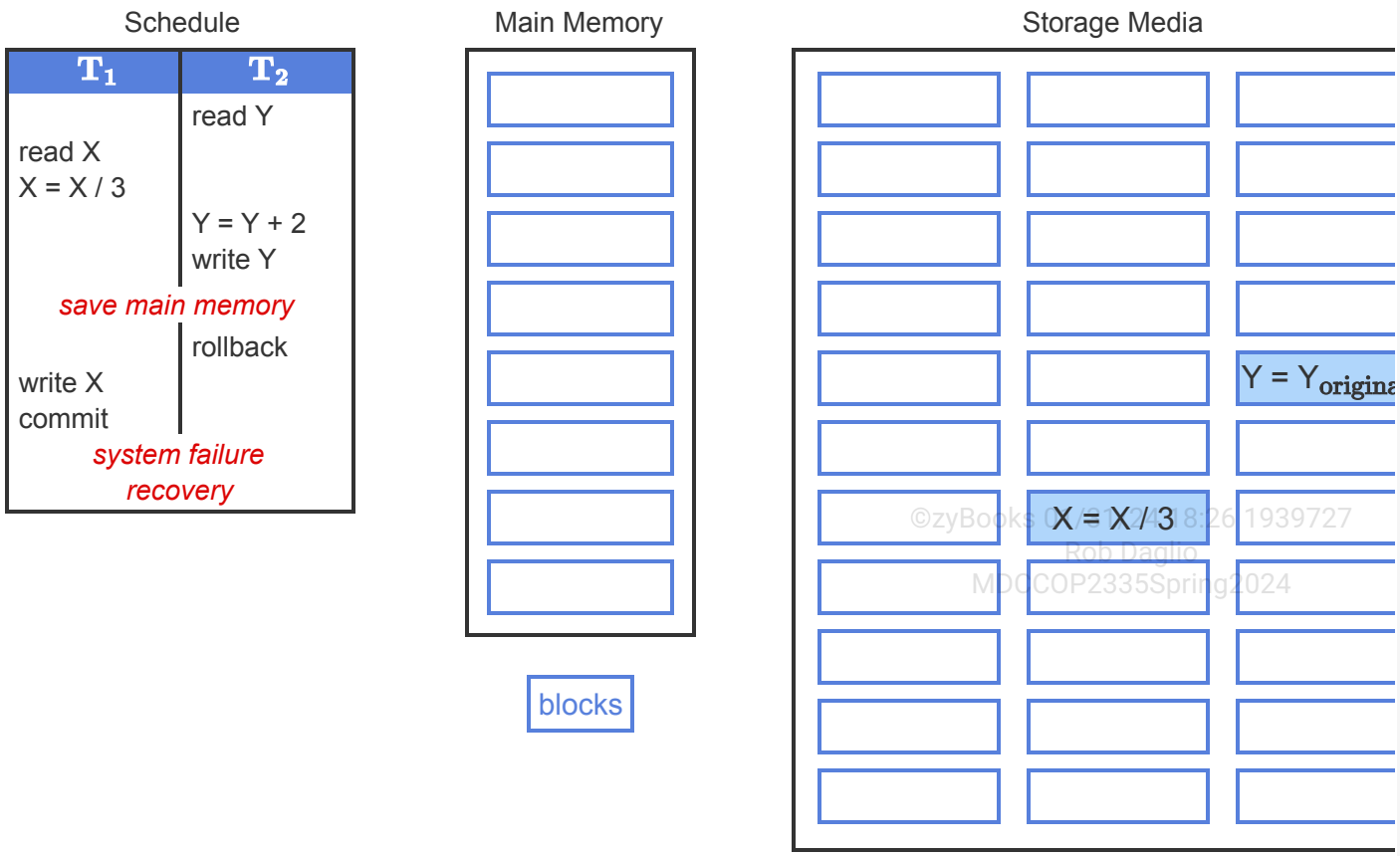
- Recovers data written to main memory, but not storage media, by committed transactions.
- Rolls back data written to storage media by uncommitted transactions.

3. A ***storage media failure*** occurs when the database is corrupted or the database connection is lost.

   Many storage systems automatically make redundant copies of data. If one copy of data is corrupted, the storage system automatically switches to a backup copy without intervention by the database or operating system. Nevertheless, storage media do fail occasionally. Alternatively, the connection between the processor and database server might fail. Either way, the database is unavailable to the application.

In principle, recovery from storage media failure is similar to recovery from system failure. In both cases, the recovery system must restore committed transactions and roll back uncommitted transactions. However, storage media failure may cause massive data loss with lengthy recovery times. Consequently, most commercial databases implement special techniques for rapid recovery from storage media failure.

PARTICIPATION ACTIVITY 39.4.1: System failure.



Animation content:

Static figure:
A schedule has actions in the following sequence. The schedule has internal captions, shown in parentheses:
T2: read Y
T1: read X
T1: X = X / 3
T2: Y = Y + 2
T2: write Y
(save main memory)
T2: rollback
T1: write X
T1: commit
(system failure)
(recovery)

A main memory diagram contains 8 blocks.

A storage media diagram contains 33 blocks. One block contains X = X / 3. Another block contains Y = Y original.

Step 1: The database processes two transactions. The schedule appears without internal captions. Main memory and storage media diagrams appear. All blocks are empty.

Step 2: The database writes a new Y value to a block in main memory. Action T2: write Y is highlighted. Y = Y + 2 appears in a main memory block.

Step 3: The database saves main memory to storage media. All saved blocks in main memory are released for reuse. The save main memory caption appears in the schedule. Y = Y + 2 moves from the main memory block to a storage media block.

Step 4: The database rolls back T2, restoring Y's original value in main memory. Action T2: rollback is highlighted. Y = Y original appears in a main memory block.

Step 5: The database writes a new X value in main memory and commits T1. Action T1: write X is highlighted. X = X / 3 appears in a main memory block.

Step 6: Before main memory is saved to storage media, the operating system fails, and main memory is lost. The system failure caption appears in the schedule. All text in main memory blocks disappears.

Step 7: The recovery system must restore 'X = X / 3' and roll back 'Y = Y + 2' in storage media. The recovery caption appears in the schedule. Y = Y original appears in a storage media block. X = X / 3 appears in another storage media block.

Animation captions:

1. The database processes two transactions.
2. The database writes a new Y value to a block in main memory.
3. The database saves main memory to storage media. All saved blocks in main memory are released for reuse.
4. The database rolls back T_2, restoring Y's original value in main memory.
5. The database writes a new X value in main memory and commits T_1.
6. Before main memory is saved to storage media, the operating system fails, and main memory is lost.
7. [illegible]

PARTICIPATION ACTIVITY 39.4.2: Failure scenarios.

Select the failure scenario that best describes the example.

1) Application programs run on a client machine. The database runs on a separate server machine. The network between client and server fails, and the database does not respond to any application requests.
   - ○ Transaction failure
   - ○ System failure
   - ○ Storage media failure

2) The database detects two deadlocked transactions. To break the deadlock, the database rolls back one of the transactions.
   - ○ Transaction failure
   - ○ System failure
   - ○ Storage media failure

3) Due to a security breach, computer memory is corrupted. After the security breach is resolved, the database administrator restarts the database.

- ○ Transaction failure
- ○ System failure
- ○ Storage media failure

Recovery log

A ***recovery log*** is a file containing a sequential record of all database operations. The log and database are stored on different storage media so the log survives database failures. The recovery system uses the log to restore the database after a failure.

The log contains transaction and data identifiers. Transaction identifiers are assigned by the database system for internal use by the recovery and concurrency systems. Data identifiers correspond to table name, column name, and primary key value, but are compressed or encoded for efficiency.

The recovery log contains four types of records:

1. An ***update record*** indicates a transaction has changed data. Update records include the transaction identifier, the data identifier, the original data value, and the new data value. Update records also track insert and delete operations.
2. A ***compensation record*** , also known as an ***undo record*** , indicates data has been restored to the original value during a rollback. Compensation records include the transaction identifier, the data identifier, and the restored (original) data value. Compensation records are necessary because the log must capture every database change, including changes executed by a rollback.
3. A ***transaction record*** indicates a transaction boundary. Three types of transaction records exist: start, commit, and rollback. Transaction records include the 'start', 'commit', or 'rollback' indicator and the transaction identifier.
4. A ***checkpoint record*** indicates that all data in main memory has been saved on storage media. When the database executes a checkpoint, transaction processing is suspended while all unsaved data and log records are written to storage media. In the event of a system failure, the recovery system reads only log records following the last checkpoint, rather than the entire file. Checkpoint records include a 'checkpoint' indicator along with the identifiers of all transactions that are active (uncommitted) at the time of the checkpoint.

The above four record types provide a complete history of database operations and enable recovery from all three failure scenarios.

PARTICIPATION ACTIVITY 39.4.3: Recovery log records.

Schedule

T_1	T_2
	read Y
read X	
X = X / 3	
	Y = Y + 2
	write Y
save main memory	
	rollback
write X	
commit	

Recovery Log

start T_2
start T_1
update T_2, Y_{ID}, $Y_{original}$, Y_{new}
checkpoint T_1, T_2
undo T_2, Y_{ID}, $Y_{original}$
rollback T_2
update T_1, X_{ID}, $X_{original}$, X_{new}
commit T_1

Animation content:

Static figure:
A schedule has actions in the following sequence. The schedule has internal captions, shown in parentheses:
T2: read Y
T1: read X
T1: X = X / 3
T2: Y = Y + 2
T2: write Y
(save main memory)
T2: rollback
T1: write X
T1: commit

A recovery log has the following records:
start T2
start T1
update T2, Y ID, Y original, Y new
checkpoint T1, T2
undo T2, Y ID, Y original
rollback T2
update T1, X ID, X original, X new
commit T1

Step 1: When the transactions execute, start records are written in the recovery log. The schedule appears. An empty recovery log appears. The two read actions are highlighted. The two start records appear in the recovery log.

Step 2: When T2 writes Y, an update record is written in the log. Action write Y is highlighted. The first update record appears in the recovery log.

Step 3: When the system saves all updates from main memory to storage media, a checkpoint record is written in the log. The save main memory caption is highlighted. The checkpoint record appears in the recovery log.

Step 4: When T2 rolls back, compensation and rollback records are written in the log. Action rollback is highlighted. The undo and rollback records appear in the recovery log.

Step 5: When T1 writes X, an update record is written in the log. Action write X is highlighted. The second update record appears in the recovery log.

Step 6: When T1 commits, a commit record is written in the log. Action commit is highlighted. The commit record appears in the recovery log.

Animation captions:

1. When the transactions execute, start records are written in the recovery log.
2. When T_2 writes Y, an update record is written in the log.
3. When the system saves all updates from main memory to storage media, a checkpoint record is written in the log.
4. When T_2 rolls back, compensation and rollback records are written in the log.
5. When T_1 writes X, an update record is written in the log.
6. When T_1 commits, a commit record is written in the log.

PARTICIPATION ACTIVITY 39.4.4: Recovery log.

1) After all updates have been reversed in a rollback, a(n) _____ record is written in the log.

Check Show answer

2) When a transaction deletes a table row, a(n) _____ record is written in the log.

Check Show answer

3) A(n) _____ record indicates all data is saved from main memory to storage media.

Check Show answer

4) A(n) _____ record always appears in the log at the beginning of a transaction.

Check Show answer

5) A(n) _____ record is written in the log whenever an update is reversed during a rollback.

Check Show answer

6) A list of active transactions appears in a(n) _____ record.

Check Show answer

Recovery from transaction failure

Recovery from a transaction failure is relatively simple. A database component, such as the concurrency system, instructs the recovery system to roll back transaction T. The recovery system

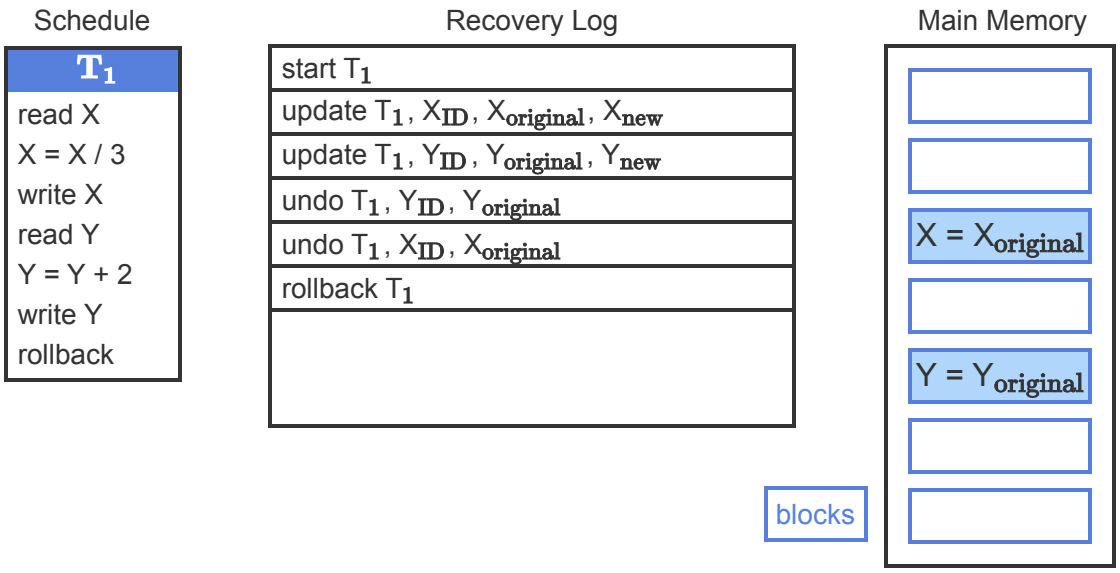
reads the recovery log *backwards*, searching for update records for T. For each update record, the recovery system:

- Restores data D to original value V in the update record.
- Writes a compensation record for T, D, and V to the end of the log.

Eventually, the recovery system reads the 'start T' transaction record and writes a 'rollback T' record to the end of the log. At this point, the rollback is complete.

The recovery system reads the log backwards because data must be restored in reverse order of transaction operations.

PARTICIPATION ACTIVITY 39.4.5: Recovery from transaction failure.



Animation content:

Static figure:
A schedule has one transaction with actions in the following sequence:
T1: read X
T1: X = X / 3
T1: write X
T1: read Y
T1: Y = Y + 2
T1: write Y
T1: rollback

A recovery log has these records:

start T1
update T1, X ID, X original, X new
update T1, Y ID, Y original, Y new
undo T1, Y ID, Y original
undo T1, X ID, X original
rollback T1

A main memory diagram has seven blocks. One block contains X = X original. Another block contains Y = Y original.

Step 1: As the transaction executes, a start record is written to the recovery log. The schedule, recovery log, and main memory appear. The recovery log and main memory are empty. The start record appears in the recovery log.

Step 2: Write statements generate update records in the log and save data in main memory. Actions write X and write Y are highlighted. The two update records appear. X = X / 3 and Y = Y + 2 appear in main memory blocks..

Step 3: When the system or programmer initiates rollback, the recovery system reads the log in reverse. Action rollback is highlighted. The second update record is highlighted.

Step 4: Y is restored to original value in main memory, and a compensation record is written to the log. The first undo record appears. In main memory, Y = Y original replaces Y = Y + 2.

Step 5: X is restored to original value in main memory, and a compensation record is written to the log. The first update record is highlighted. The second undo record appears. In main memory, X = X original replaces X = X / 3.

Step 6: The rollback record is written to the log. The rollback is complete. The rollback record appears.

Animation captions:

1. As the transaction executes, a start record is written to the recovery log.
2. Write statements generate update records in the log and save data in main memory.
3. When the system or programmer initiates rollback, the recovery system reads the log in reverse.
4. Y is restored to original value in main memory, and a compensation record is written to the log.
5. X is restored to original value in main memory, and a compensation record is written to the log.
6. The rollback record is written to the log. The rollback is complete.

PARTICIPATION ACTIVITY 39.4.6: Recovery from transaction failure.

Refer to the schedule below :

T_1 reads X
T_2 reads Y
T_2 writes Z
T_1 writes Y
T_2 writes X
deadlock

Deadlock occurs because T_1 is waiting for an exclusive lock on Y, and T_2 is waiting for an exclusive lock on X. When deadlock occurs, the database rolls back T_2 and completes T_1.

Order the log records to match the schedule.

If unable to drag and drop, refresh the page.

undo T_2, Z . . . | **start T_1** | **commit T_1** | **update T_1, Y . . .** | **start T_2** | **rollback T_2** | **update T_2, Z . . .**

	Log record 1
	Log record 2
	Log record 3
	Log record 4
	Log record 5
	Log record 6
	Log record 7

Reset

Recovery from system failure

Recovery from a system failure has two phases. The ***redo phase*** restores all transactions that were committed or rolled back since the last checkpoint. The ***undo phase*** rolls back transactions that were neither committed nor rolled back.

In the *redo phase*, the recovery system reads the recovery log *forward* from the latest checkpoint record. While reading the log, the recovery system maintains a list of active transactions. The list is initialized with active transactions in the checkpoint record. As each log record is read:

- Transactions in a 'start' transaction record are added to the list.
- Transactions in a 'commit' or 'rollback' transaction record are removed from the list.
- Data in an update record is set to the new value.
- Data in a compensation record is restored to the original value.

The above redo process performs unnecessary work on rolled-back transactions. The process changes data in update records and later restores the same data in compensation records. In principle, the recovery system could avoid unnecessary work by ignoring all rolled-back transactions. Processing all log records is simpler than ignoring rolled-back transactions, however, and is thus commonly implemented.

At the end of the redo phase, all transactions remaining on the list have no 'commit' or 'rollback' records. These transactions are rolled back in the undo phase.

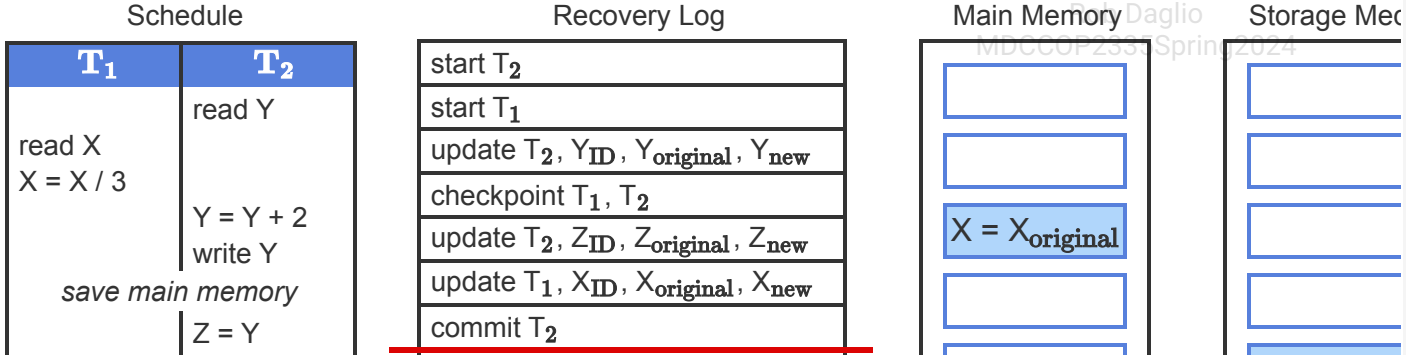
During the *undo phase*, the recovery system reads the recovery log *backwards* from the last record. While reading the log, the recovery system maintains a list of incomplete transactions. The list is initialized with active transactions remaining from the redo phase. Each transaction on the list is rolled back:

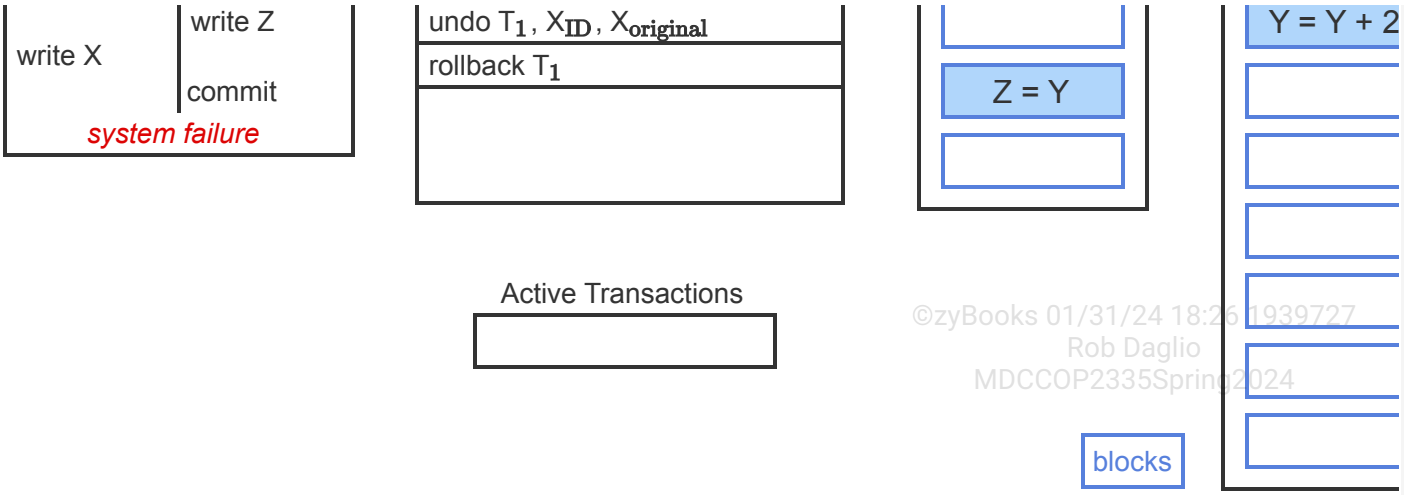
- When an update record is read, data is restored to the original value, and a compensation record is written at the end of the log.
- When the 'start' transaction record is read, the transaction is removed from the list, and a 'rollback' transaction record is written at the end of the log.

The above process is the same as the rollback process for transaction failures, described above.

When the transaction list is empty, the undo phase ends, and recovery is complete.

PARTICIPATION ACTIVITY 39.4.7: Recovery from system failure.





Animation content:

Static figure:
A schedule has actions in the following sequence. The schedule has internal captions, shown in parentheses:
T2: read Y
T1: read X
T1: X = X / 3
T2: Y = Y + 2
T2: write Y
(save main memory)
T2: Z = Y
T2: write Z
T1: write X
T2: commit
(system failure)

A recovery log has the following records:
start T2
start T1
update T2, Y ID, Y original, Y new
checkpoint T1, T2
update T2, Z ID, Z original, Z new
update T1, X ID, X original, X new
commit T2
undo T1, X ID, X original
rollback T1

A red line appears below the commit record. An empty list of active transactions appears under the recovery log.

A main memory diagram contains 7 blocks. One block contains X = X original. Another block contains Z = Y

A storage media diagram has 11 blocks. One block contains Y = Y + 2.

Step 1: The schedule executes normally. The checkpoint record indicates transactions T1 and T2 are active. The schedule, recovery log, main memory, and storage media appear. The schedule executes through the commit action. All records through commit appear in the recovery log. X = X / 3 and Z = Y appear in main memory blocks. Y = Y + 2 appears in a storage media block.

Step 2: The system fails, and memory is lost. The caption system failure appears in the schedule. A red line appears under the commit record. Main memory blocks are erased.

Step 3: The recovery system initializes the list of active transactions from the last checkpoint. The checkpoint record is highlighted. T1 and T2 appear in the active transactions list.

Step 4: During the redo phase, the system reads the log forward from the checkpoint and restores data from update records. The two update records are highlighted. X = X / 3 and Z = Y reappear in main memory blocks.

Step 5: The commit record in the log removes T2 from the transaction list. The commit T2 record is highlighted. T2 disappears from the active transactions record.

Step 6: During the undo phase, the system reads the log backwards and rolls back changes made by transactions in the list. The undo record appears. The update T1 record is highlighted. X = X original replaces X = X / 3 in main memory.

Step 7: The system reads the 'start T1' record in the log, writes a rollback record and deletes T1 from the list. Recovery is complete. The start T1 record is highlighted. The rollback T1 record appears. T1 disappears from the active transactions record.

Animation captions:

1. The schedule executes normally. The checkpoint record indicates transactions T_1 and T_2 are active.
2. The system fails, and memory is lost.
3. The recovery system initializes the list of active transactions from the last checkpoint.
4. During the redo phase, the system reads the log forward from the checkpoint and restores data from update records.
5. The commit record in the log removes T_2 from the transaction list.
6. During the undo phase, the system reads the log backwards and rolls back changes made by transactions in the list.
7. The system reads the 'start T_1' record in the log, writes a rollback record and deletes T_1 from the list. Recovery is complete.

PARTICIPATION ACTIVITY 39.4.8: Recovery from system failure.

1) During the *redo* phase, what log records remove a transaction from the active transaction list?

- Commit records only
- Rollback records only
- Both commit and rollback records

2) During the *redo* phase, what log records generate a database write?

- Update records only
- Compensation (undo) records only
- Both update and compensation (undo) records

3) During the *undo* phase, the recovery system reads the log in reverse and stops at:

- The most recent checkpoint record
- The start record for the last transaction in the active transaction list
- The first start record after the most recent checkpoint

4) During the *undo* phase, the recovery system writes compensation records for:

- Transactions that do not commit or roll back following the most recent checkpoint
- All transactions listed in the most recent checkpoint
- All transactions that roll back following the most recent checkpoint

CHALLENGE ACTIVITY 39.4.1: Recovery.

539740.3879454.qx3zqy7

Start

Given the schedule below, complete the recovery log to match the schedule.

Schedule

T_1	T_2
	read Y
read X	
X = X / 8	
	Y = Y + 7
	write Y

save main memory

	rollback
write X	
commit	

Recovery Log

start T_2
start T_1
update T_2, Y_{ID}, $Y_{original}$, Y_{new}
(A)
undo T_2, Y_{ID}, $Y_{original}$
(B)
(C)
commit T_1

(A) Pick

(B) Pick

(C) Pick

2 3 4

Check Next

Recovery from storage media failure

Availability is the percentage of time a system is working from the perspective of the system user. Many databases require high availability, in excess of 99% of the time. Ex: A database that manages stock exchange trades must have availability approaching 100%.

Availability is a primary concern in recovery from storage media failures.

In principle, recovery from storage media failure might be the same as recovery from system failure, starting at the beginning of the recovery log. The log contains all database operations, so the database may be fully restored from scratch. In practice, however, this approach is not feasible. The log of all database history is exceedingly large, so recovery time is exceedingly long and availability exceedingly low.

Two recovery techniques are commonly used: cold backup and hot backup.

The ***cold backup*** technique periodically creates checkpoints and, while transaction processing is paused, copies the database to backup media. The backup media might be tapes or disk drives, normally stored in a separate location from the primary database. Typically, checkpoints and backups are taken hourly or daily when database activity is low.

When storage media fails, the recovery system:

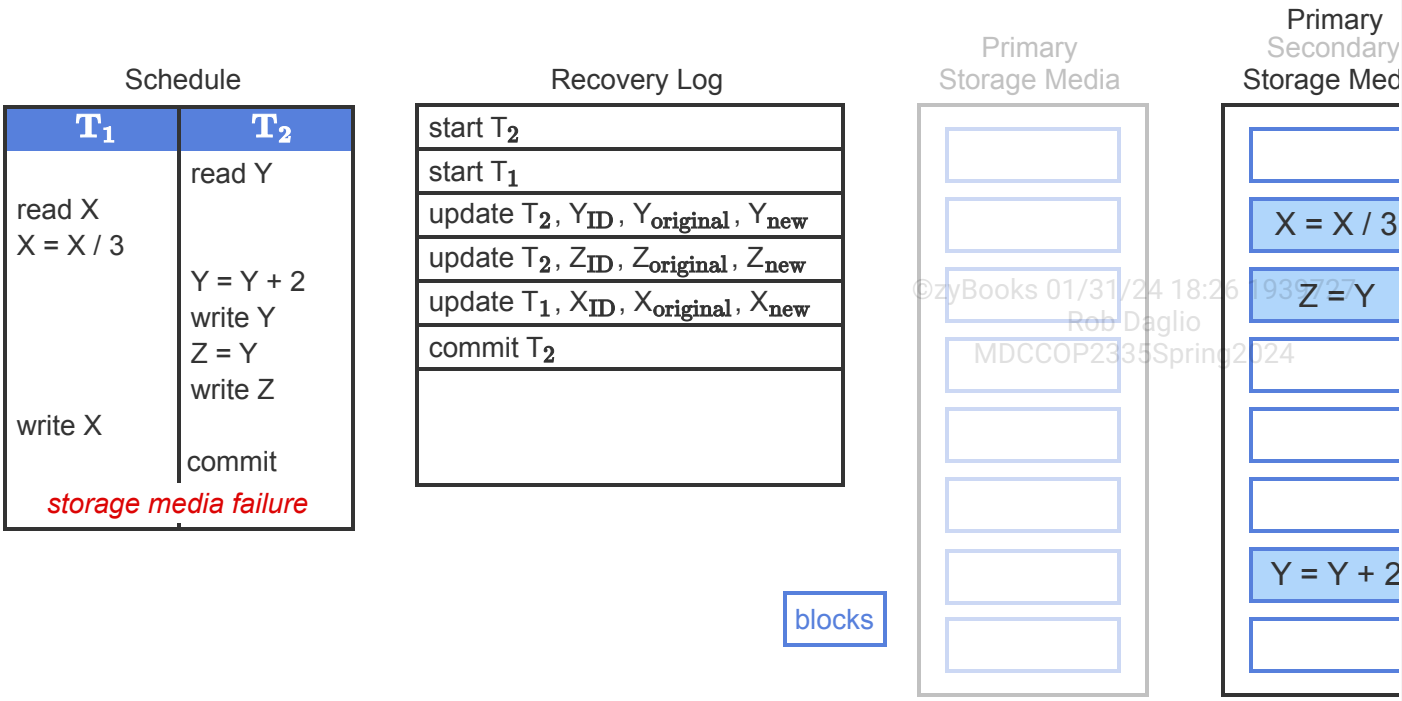
1. Copies the latest backup to the database.
2. Executes the system failure recovery process beginning at the latest checkpoint.

Depending on the backup size and period length, recovery might require minutes or hours. The database is unavailable during recovery, so the cold backup technique is used when low availability is acceptable.

The ***hot backup*** technique maintains a secondary database that is nearly synchronized with the primary database. As the primary database processes transactions, log records are sent to and processed by the secondary database. If the databases share a high-speed connection, the secondary database is only moments behind the primary database.

When storage media for the primary database fails, the secondary database becomes primary. This can be accomplished, for example, by swapping the internet addresses of primary and secondary servers. The new primary database is available in a matter of seconds, as soon as outstanding log records are processed.

PARTICIPATION ACTIVITY 39.4.9: Hot backup technique.



Animation content:

Static figure:
A schedule has actions in the following sequence. The schedule has internal captions, shown in parentheses:
T2: read Y
T1: read X
T1: X = X / 3
T2: Y = Y + 2
T2: write Y
T2: Z = Y
T2: write Z
T1: write X
T2: commit
(storage media failure)

A recovery log has the following records:
start T2
start T1
update T2, Y ID, Y original, Y new
update T2, Z ID, Z original, Z new
update T1, X ID, X original, X new
commit T2

A primary storage media diagram has empty blocks and is faded out.

A secondary storage media diagram appears. The following actions appear in three blocks:
X = X / 3
Z = Y
Y = Y + 2

Step 1: As transactions are executed, log records are sent to a secondary database. The schedule appears. Action T2: write Y is highlighted. The recovery log appears with only the start records and the first update record. Y = Y + 2 appears in blocks of both primary and secondary storage media.

Step 2: The secondary database may be in a remote location. Updates to the secondary database may be slightly delayed. Actions write Z, write X, and commit are highlighted. The remaining records appear in the recovery log. X = X / 3 and Z = Y appear in primary storage media blocks.

Step 3: The primary database fails. Caption storage media failure is highlighted in the schedule. Primary storage media is erased.

Step 4: After a momentary delay, the secondary database receives the remaining log records. X = X / 3 and Z = Y appear in secondary storage media blocks.

Step 5: The secondary database is now synchronized and becomes the primary database. The caption of secondary storage media changes to primary storage media.

Animation captions:

1. As transactions are executed, log records are sent to a secondary database.
2. The secondary database may be in a remote location. Updates to the secondary database may be slightly delayed.
3. The primary database fails.
4. After a momentary delay, the secondary database receives the remaining log records.
5. The secondary database is now synchronized and becomes the primary database.

PARTICIPATION ACTIVITY 39.4.10: Recovery from storage media failure.

1) Database availability is:

- The percentage of time that a database is working properly.
- The number of seconds per day that a database is responsive to application programs.
- The percentage of time that a database is responsive to application programs.

2) With a cold backup, recovery from storage media failure reads the log:

- From the beginning.
- From the latest checkpoint.
- From the latest commit record.

3) With a hot backup, as a transaction executes against the primary database:

- Log records are sent to the secondary database.
- The secondary database is updated synchronously, when the transaction commits or rolls back.
- The operating system synchronizes the secondary database.

39.5 Transactions with SQL

Isolation levels

Transaction statements are standardized in SQL and implemented in most relational databases. However, transaction statements interact with concurrency and recovery systems, which vary significantly. Consequently, most databases offer many nonstandard extensions.

This section describes transaction statements for MySQL configured with the InnoDB storage engine. The statements are similar for other storage engines and databases. MySQL storage engines are

described elsewhere in this material.

The ***SET TRANSACTION*** statement sets the isolation level for subsequent transactions:

Figure 39.5.1: SET TRANSACTION syntax.

```
SET [ GLOBAL | SESSION ] TRANSACTION
ISOLATION LEVEL [ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED ];
```


The keyword following ISOLATION LEVEL balances isolation of concurrent transactions against performance. SERIALIZABLE provides complete isolation with longer transaction duration. READ UNCOMMITTED allows conflicts between concurrent transactions with shorter duration. The four isolation levels are described elsewhere in this material.

The GLOBAL and SESSION keywords determine the scope of the isolation level:

- ***SESSION*** sets the isolation level for all transactions in the current session. A ***session*** is a series of SQL statements submitted to a MySQL server, beginning when a user or program connects to the server and ending when the user or program disconnects.
- ***GLOBAL*** sets the isolation level for all transactions submitted to the MySQL server for all subsequent sessions. Existing sessions are not affected. The GLOBAL setting can be used only by a database administrator with appropriate MySQL privileges.

When neither keyword is specified, the isolation level applies only to the next transaction in the session.

PARTICIPATION ACTIVITY 39.5.1: SET TRANSACTION statement.

Isolation Level	Operation
	session begins `SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;` *session ends*
REPEATABLE READ	*session begins* transaction 1 `SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;`

SERIALIZABLE transaction 2
SERIALIZABLE transaction 3

```
SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;
```

READ COMMITTED transaction 4
SERIALIZABLE transaction 5
session ends

Animation content:

Static figure:
A mix of pseudocode and SQL appears. Each transaction has a caption, which appears in parentheses.
Begin SQL code:
session begins
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;
session ends
session begins
transaction 1 (REPEATABLE READ)
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
transaction 2 (SERIALIZABLE)
transaction 3 (SERIALIZABLE)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
transaction 4 (READ COMMITTED)
transaction 5 (SERIALIZABLE)
session ends
End SQL code.

Step 1: The database administrator sets the global isolation level to REPEATABLE READ. The first two code lines appear:
session begins
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;

Step 2: Global isolation level takes effect in new sessions. Transaction 1 uses REPEATABLE READ isolation level. The next three code lines appear:
session ends
session begins
transaction 1 (REPEATABLE READ)

Step 3: The programmer sets session isolation level to SERIALIZABLE. The next code line appears:
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Step 4: Session level overrides global level for the remainder of the session. Transactions 2 and 3 use SERIALIZABLE isolation level. The next two code lines appear:
transaction 2 (SERIALIZABLE)
transaction 3 (SERIALIZABLE)

Step 5: SET TRANSACTION overrides the session level for the next transaction. Transaction 4 uses READ COMMITTED isolation level. The next two code lines appear:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
transaction 4 (READ COMMITTED)

Step 6: Subsequent transactions revert to session level. Transaction 5 uses SERIALIZABLE isolation level. The remaining code lines appear:
transaction 5 (SERIALIZABLE)
session ends

Animation captions:

1. The database administrator sets the global isolation level to REPEATABLE READ.
2. Global isolation level takes effect in new sessions. Transaction 1 uses REPEATABLE READ isolation level.
3. The programmer sets session isolation level to SERIALIZABLE.
4. Session level overrides global level for the remainder of the session. Transactions 2 and 3 use SERIALIZABLE isolation level.
5. SET TRANSACTION overrides the session level for the next transaction. Transaction 4 uses READ COMMITTED isolation level.
6. Subsequent transactions revert to session level. Transaction 5 uses SERIALIZABLE isolation level.

PARTICIPATION ACTIVITY 39.5.2: SET TRANSACTION statement.

Refer to the following sequence:

session begins

```
SET GLOBAL TRANSACTION
ISOLATION LEVEL READ UNCOMMITTED;
```

session ends

session begins

```
SET TRANSACTION
ISOLATION LEVEL SERIALIZABLE;
```

transaction 1
transaction 2

```
SET SESSION TRANSACTION
ISOLATION LEVEL REPEATABLE READ;
```

transaction 3

transaction 4

```
SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;
```

transaction 5

session ends

Match the isolation level to the transaction.

If unable to drag and drop, refresh the page.

READ UNCOMMITTED | SERIALIZABLE | READ COMMITTED | REPEATABLE READ

	Transaction 1
	Transaction 2
	Transaction 4
	Transaction 5

Reset

Transaction boundaries

A ***transaction boundary*** is the first or last statement of a transaction. A transaction boundary is one of three statements:

- The ***START TRANSACTION*** statement starts a new transaction.
- The ***COMMIT*** statement commits the current transaction.
- The ***ROLLBACK*** statement rolls back the current transaction.

All statements between START TRANSACTION and the following COMMIT or ROLLBACK comprise a single transaction.

Figure 39.5.2: Transaction statements.

```
START
TRANSACTION;

COMMIT;

ROLLBACK;
```


In MySQL with InnoDB, the behavior of statements outside a transaction boundary depends on the ***autocommit*** system variable. When autocommit is ON, individual statements are separate transactions and immediately commit or roll back. When autocommit is OFF, individual statements do not commit immediately. Instead, a new transaction begins automatically when a program starts and after each COMMIT and ROLLBACK. The default setting is ON.

System variables like autocommit are assigned a value using a ***SET*** statement, as shown in the figure below.

Figure 39.5.3: SET autocommit.

```
SET autocommit = [ OFF | ON
];
```

COMMIT and ROLLBACK have optional keywords:

- ***AND CHAIN*** overrides the autocommit setting and starts a new transaction, as if a START TRANSACTION were executed. The isolation level of the new transaction is the same as the prior transaction.
- ***RELEASE*** ends the current session and disconnects from the server.

Figure 39.5.4: COMMIT and ROLLBACK statements.


```
COMMIT [ AND CHAIN ] [ RELEASE ];

ROLLBACK [ AND CHAIN ] [ RELEASE
];
```

Data definition statements such as CREATE TABLE and DROP INDEX, and administrative statements such as ANALYZE TABLE, always commit immediately. These statements behave as if a COMMIT were executed before and after the statement.

BEGIN statement

*The **BEGIN** statement is identical to START TRANSACTION. However, BEGIN is easily confused with the BEGIN - END statement pair, which has different meaning in many programming languages. For this reason, the BEGIN statement is not recommended.*

PARTICIPATION ACTIVITY 39.5.3: Transaction boundary statements.

Transaction Boundary	Operation
	session begins `SET autocommit = ON;` *session ends*
	session begins
start and end	`UPDATE ... ;`

start	`START TRANSACTION;`
	`INSERT ... ;`
	`SELECT ... ;`
	`UPDATE ... ;`

end	`COMMIT AND CHAIN;`
start	`UPDATE ... ;` `UPDATE ... ;`
end	`ROLLBACK;`
start and end	`DELETE ... ;` *session ends*

Animation content:

Static figure:
A mix of pseudocode and SQL appears. Some code lines have captions. Captions indicate transaction boundaries and appear in parentheses.

Begin SQL code:
session begins
SET autocommit = ON;
session ends
session begins
UPDATE . . . ; (start and end)
START TRANSACTION; (start)
INSERT . . . ;
SELECT . . . ;
UPDATE . . . ;
COMMIT AND CHAIN; (end)
UPDATE . . . ; (start)
UPDATE . . . ;
ROLLBACK; (end)
DELETE . . . ; (start and end)
session ends
End SQL code.

Step 1: The data administrator sets system autocommit to ON. The first three code lines appear:
session begins
SET autocommit = ON;
session ends

Step 2: Since autocommit is ON, the UPDATE statement commits immediately. The next two code lines appear:
session begins
UPDATE . . . ; (start and end)

Step 3: START TRANSACTION overrides autocommit. The transaction continues until the COMMIT statement.
The next five code lines appear. Keywords AND CHAIN do not appear in the last line:
START TRANSACTION; (start)
INSERT . . . ;
SELECT . . . ;
UPDATE . . . ;
COMMIT; (end)

Step 4: AND CHAIN starts a new transaction. Keyword AND CHAIN is added to the COMMIT statement. The next three code lines appear:
UPDATE . . . ; (start)
UPDATE . . . ;
ROLLBACK; (end)

Step 5: Neither AND CHAIN nor START TRANSACTION appear, so the DELETE statement autocommits. The remaining code lines appear:
DELETE . . . ; (start and end)
session ends

Animation captions:

1. The data administrator sets system autocommit to ON.
2. Since autocommit is ON, the UPDATE statement commits immediately.
3. START TRANSACTION overrides autocommit. The transaction continues until the COMMIT statement.
4. AND CHAIN starts a new transaction.
5. Neither AND CHAIN nor START TRANSACTION appear, so the DELETE statement autocommits.

PARTICIPATION ACTIVITY 39.5.4: Transaction boundaries.

Refer to the following sequence:

session begins

```
SET autocommit = OFF;
```

session ends

session begins
statement 1
statement 2
statement 3

```
START TRANSACTION;
```

statement 4

```
COMMIT AND CHAIN;
```

statement 5
statement 6
statement 7

```
ROLLBACK;
```

statement 8
session ends

Match the transaction boundary to the statement.

If unable to drag and drop, refresh the page.

start | neither start nor end | both start and end | end

	Statement 1
	Statement 3
	Statement 6
	Statement 8

Reset

Savepoints

A ***savepoint*** is a point within a transaction where partial transaction results are saved temporarily. Savepoints prevent redoing work in lengthy transactions. If an error occurs within a transaction after a savepoint, the transaction can restart at the savepoint rather than at the beginning.

Savepoints are managed with three statements using an identifier, the savepoint name:

- The ***SAVEPOINT*** statement saves internal transaction data and associates the data with the identifier.
- The ***RELEASE SAVEPOINT*** statement discards the identifier and saved data.
- The ***ROLLBACK TO*** statement resets transaction data to the savepoint values, restarts processing at the savepoint, and releases all subsequent savepoints.

Savepoints are temporary and internal to a transaction. When the transaction commits, all savepoints are released.

Figure 39.5.5: Savepoint statements.

```
SAVEPOINT identifier;

ROLLBACK TO identifier;

RELEASE SAVEPOINT
identifier;
```


PARTICIPATION ACTIVITY

39.5.5: SAVEPOINT statement.

Employee

ID	Name	Salary
5384	Sam Snead	30500
6381	Maria Rodriguez	92300
8820	Jiho Chen	51000

Operation

session begins

```
START TRANSACTION;
SELECT * FROM Employee;

DELETE FROM Employee WHERE Name = 'Lisa Ellison
SELECT * FROM Employee;
SAVEPOINT First;

DELETE FROM Employee WHERE Name = 'Sam Snead';
DELETE FROM Employee WHERE name = 'Jiho Chen';
SELECT * FROM Employee;
SAVEPOINT Second;

ROLLBACK TO First;
SELECT * FROM Employee;

COMMIT;
```

session ends

Animation content:

Static figure:
The Employee table has columns ID, Name, and Salary. ID is the primary key. Employee has three rows:
5384, Sam Snead, 30500
6381, Maria Rodriguez, 92300
8820, Jiho Chen, 51000

A mix of pseudocode and SQL appears.
Begin SQL code:
session begins
START TRANSACTION;
SELECT * FROM Employee;
DELETE FROM Employee WHERE Name = 'Lisa Ellison';
SELECT * FROM Employee;
SAVEPOINT First;
DELETE FROM Employee WHERE Name = 'Sam Snead';
DELETE FROM Employee WHERE name = 'Jiho Chen';
SELECT * FROM Employee;
SAVEPOINT Second;

ROLLBACK TO First;
SELECT * FROM Employee;
COMMIT;
session ends
End SQL code.

Step 1: When the transaction starts, Employee has four rows. The first three code lines appear:
session begins
START TRANSACTION;
SELECT * FROM Employee;
The Employee table appears with four rows:
2538, Lisa Ellison, 45000
5384, Sam Snead, 30500
6381, Maria Rodriguez, 92300
8820, Jiho Chen, 51000

Step 2: At savepoint First, one row has been deleted from Employee. The next three code lines appear:
DELETE FROM Employee WHERE Name = 'Lisa Ellison';
SELECT * FROM Employee;
SAVEPOINT First;
The row containing Lisa Ellison disappears from the table.

Step 3: At savepoint Second, three rows have been deleted from Employee. The next four code lines appear:
DELETE FROM Employee WHERE Name = 'Sam Snead';
DELETE FROM Employee WHERE name = 'Jiho Chen';
SELECT * FROM Employee;
SAVEPOINT Second;
The rows containing Sam Snead and Jiho Chen disappear from the table.

Step 4: Rollback restores Employee to savepoint First. All savepoint identifiers and partial transaction data are released. The next two code lines appear:
ROLLBACK TO First;
SELECT * FROM Employee;
Rows containing Sam Snead, Maria Rodriguez, and Jiho Chen appear in the table.

Step 5: Transaction commits. Employee table is saved with three rows. The remaining code lines appear:
COMMIT;
session ends

Animation captions:

1. When the transaction starts, Employee has four rows.
2. At savepoint First, one row has been deleted from Employee.
3. At savepoint Second, three rows have been deleted from Employee.
4. Rollback restores Employee to savepoint First. All savepoint identifiers and partial transaction data are released.
5. Transaction commits. Employee table is saved with three rows.

PARTICIPATION ACTIVITY 39.5.6: Savepoints.

1) A _____ statement temporarily saves data read and written by a transaction.

Check Show answer

2) A _____ statement erases saved data for zero, one, or many savepoints.

Check Show answer

3) A _____ statement erases saved data for exactly one savepoint.

Check Show answer

4) A _____ statement reverses all changes made by a transaction.

Check Show answer

Checkpoints

A ***dirty block*** is a database block that has been updated in main memory but not yet saved on storage media. A ***checkpoint*** saves dirty blocks and log records, as follows:

1. Suspend database processing.
2. Write all unsaved log records to the log file.
3. Write all dirty blocks to storage media.
4. Write a checkpoint record in the log.
5. Resume database processing.

Checkpoints enable rapid recovery from system failure, as described elsewhere in this material.

A ***fuzzy checkpoint*** resumes processing while saving dirty blocks. A fuzzy checkpoint improves database availability but complicates recovery if the system fails while saving dirty blocks.

Checkpoint syntax and procedures are dependent on database internals and vary greatly across databases. Most relational databases have a default process for automatic checkpoints. Database administrators with appropriate privileges can modify the default process or force a checkpoint manually.

MySQL with InnoDB executes a fuzzy checkpoint every second. A database administrator can modify the default behavior by setting system variables. Ex:

- `SET flush_time = 2` changes the default interval to 2 seconds.
- `SET flush = ON` executes a fuzzy checkpoint after each SQL statement.

Log records and dirty blocks can be explicitly flushed with the ***FLUSH*** statement. Ex:

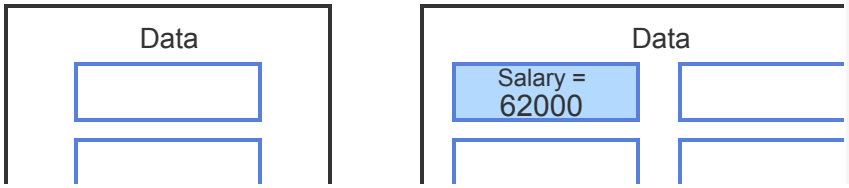
- `FLUSH TABLES` writes all dirty blocks to storage media.
- `FLUSH LOGS` saves all log records to storage media.

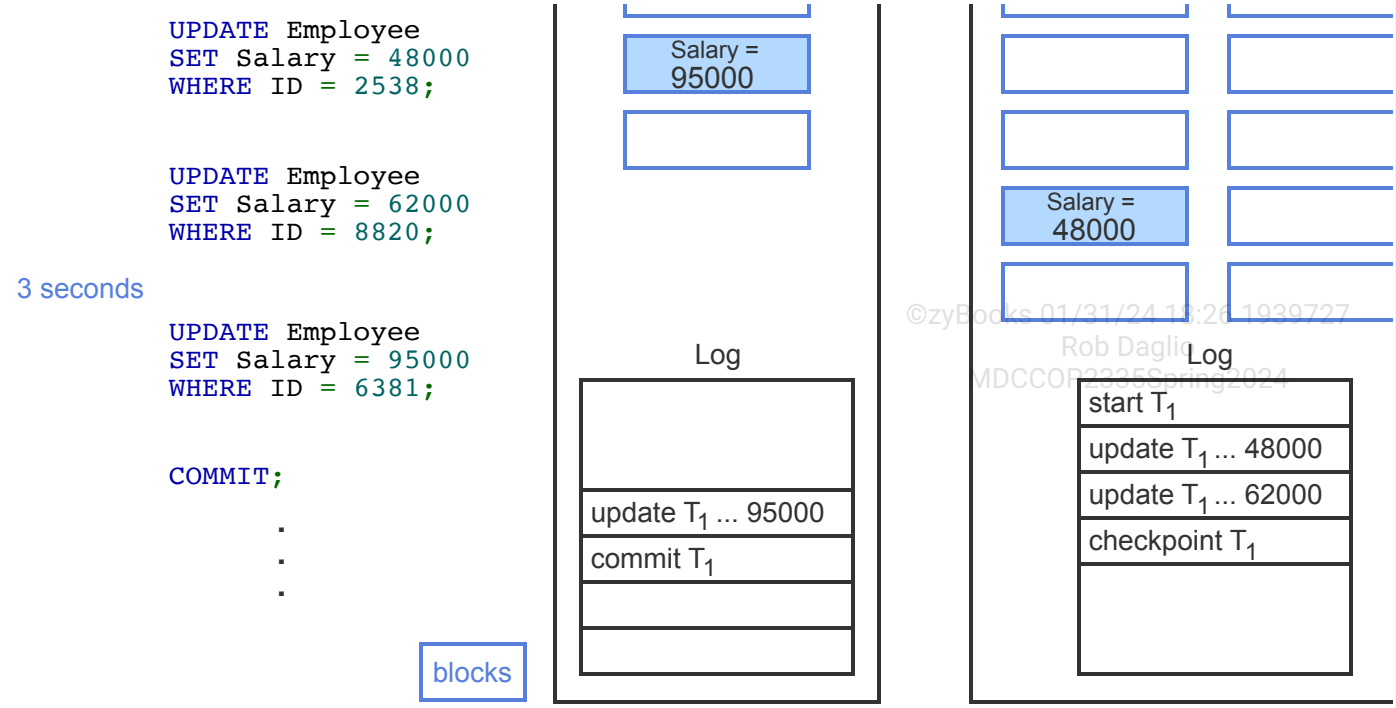
Some databases support manual checkpoints with the ***CHECKPOINT*** keyword. Ex: In SQL Server, `CHECKPOINT` forces a checkpoint. In Oracle Database, `ALTER SYSTEM CHECKPOINT` forces a checkpoint.

The above statements are examples. Commercial databases provide numerous checkpoint mechanisms with differing syntax and capabilities.

PARTICIPATION ACTIVITY 39.5.7: Checkpoints.

Time	Operations
	session begins
0 seconds	`SET flush_time = 3;`
	`START TRANSACTION;`





Animation content:

Static figure:
A mix of pseudocode and SQL appears. Some code lines have captions. Captions indicate elapsed time and appear in parentheses.
Begin SQL code:
session begins
SET flush_time = 3;
(0 seconds)
START TRANSACTION;
UPDATE Employee SET Salary = 48000 WHERE ID = 2538;
UPDATE Employee SET Salary = 62000 WHERE ID = 8820;
(3 seconds)
UPDATE Employee SET Salary = 95000 WHERE ID = 6381;
COMMIT;
End SQL code.

A main memory diagram appears. One block contains Salary = 95000. A recovery log appears inside main memory with records:
update T1 . . . 95000
commit T1

A storage media diagram appears. One block contains Salary = 62000. Another block contains Salary = 48000. A recovery log appears inside storage media with records:
start T1

update T1 . . . 48000
update T1 . . . 62000
checkpoint T1

Step 1: The data administrator configures checkpoints at three-second intervals. The first two code lines appear with caption:
session begins
SET flush_time = 3;
(0 seconds)

Step 2: Updates and log records are written to main memory. The next three code lines appear:
START TRANSACTION;
UPDATE Employee SET Salary = 48000 WHERE ID = 2538;
UPDATE Employee SET Salary = 62000 WHERE ID = 8820;
In main memory, actions appear in two blocks:
Salary = 48000
Salary = 62000
In the main memory recovery log, three records appear:
start T1
update T1 . . . 48000
update T1 . . . 62000

Step 3: After three seconds, automatic checkpoint flushes dirty blocks and log records to storage media. In the pseudocode, the caption 3 seconds appears. Data moves from main memory blocks to storage media blocks:
Salary = 48000
Salary = 62000
Recovery log records move from main memory to storage media:
start T1
update T1 . . . 48000
update T1 . . . 62000
A fourth record is added to storage media recovery log:
checkpoint T1

Step 4: The transaction continues writing data to main memory. The remaining code lines appear:
UPDATE Employee SET Salary = 95000 WHERE ID = 6381;
COMMIT;

In main memory, Salary = 95000 appears in one block. In the main memory recovery log, two records appear:
update T1 . . . 95000
commit T1

Animation captions:

1. The data administrator configures checkpoints at three-second intervals.
2. Updates and log records are written to main memory.
3. After three seconds, automatic checkpoint flushes dirty blocks and log records to storage media.
4. The transaction continues writing data to main memory.

PARTICIPATION ACTIVITY 39.5.8: Checkpoints.

1) CHECKPOINT statement syntax is specified in the SQL standard.
   - True
   - False

2) Checkpoints can be initiated either manually by the database administrator or automatically by the database.
   - True
   - False

3) A dirty block is a block that has been corrupted and cannot be read.
   - True
   - False

4) MySQL with InnoDB suspends processing during a checkpoint and restarts after a checkpoint record is written to the log.
   - True
   - False

CHALLENGE ACTIVITY 39.5.1: Transactions with SQL.

539740.3879454.qx3zqy7

Start

```
session begins
```

```
SET GLOBAL TRANSACTION
ISOLATION LEVEL READ COMMITTED;
session ends

session begins
SET TRANSACTION
ISOLATION LEVEL SERIALIZABLE;
transaction 1
SET SESSION TRANSACTION
ISOLATION LEVEL READ UNCOMMITTED;
transaction 2
transaction 3
SET TRANSACTION
ISOLATION LEVEL REPEATABLE READ;
transaction 4
transaction 5
session ends
```

Select each transaction's isolation level.

transaction 1 Pick

transaction 2 Pick

transaction 3 Pick

transaction 4 Pick

transaction 5 Pick

1 2

Check Next

Exploring further:

- MySQL transaction statements
- MySQL with InnoDB locking
- MySQL with InnoDB checkpoints

39.6 LAB - Rollback and savepoint (Sakila)

Refer to the `actor` table of the Sakila database. The table in this lab has the same columns and data types but fewer rows.

Start a transaction and:

1. Insert a new actor with values 999, 'NICOLE', 'STREEP', '2021-06-01 12:00:00'
2. Set a SAVEPOINT.
3. Delete the actor with first name 'CUBA'.

4. Select all actors.
5. Roll back to the savepoint.
6. Select all actors a second time.

The actor with first name 'CUBA' should appear in the second SELECT but not the first.

NOTE: In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.

LAB ACTIVITY | 39.6.1: LAB - Rollback and savepoint (Sakila) | 0 / 10

Main.sql — Load default template...

```
1 -- Your SQL statements go here
```

Develop mode | **Submit mode**

Explore the database and run your program as often as you'd like, before submitting for grading. Click **Run program** and observe the program's output in the second box.

Run program

Main.sql (Your program) → Output (shown below)

Program output displayed here

Coding trail of your work [What is this?](#)