Databases—Inside the Blackbox

COMPSCI 2DB3: Databases

Jelle Hellings   Holly Koponen

Department of Computing and Software
McMaster University

Winter 2024

Recap

- Modeling data.
- Querying data.
- Defining data and constraints.
- Reasoning with data constraints.

Recap

- Modeling data.
- Querying data.
- Defining data and constraints.
- Reasoning with data constraints.

The final steps

A brief overview of how a database operates.
Main focus: Concurrency Control.

A note on the book

The book does show its age a bit after almost two decades…

A note on the book

The book does show its age a bit after almost two decades…

A lot has changed

Hardware	CPUs, memory, storage, ….
Environment	Networking, “the cloud”, ….
Data	Volumes, computations, ….

A note on the book

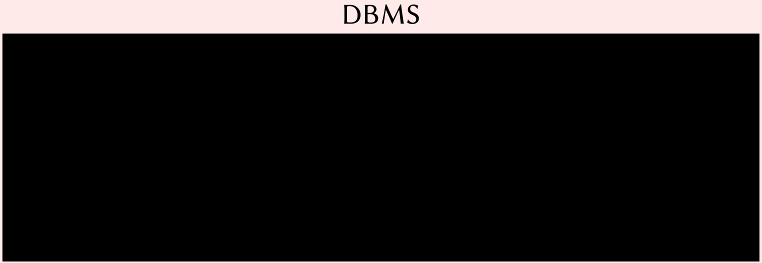
The book does show its age a bit after almost two decades…

A lot has changed

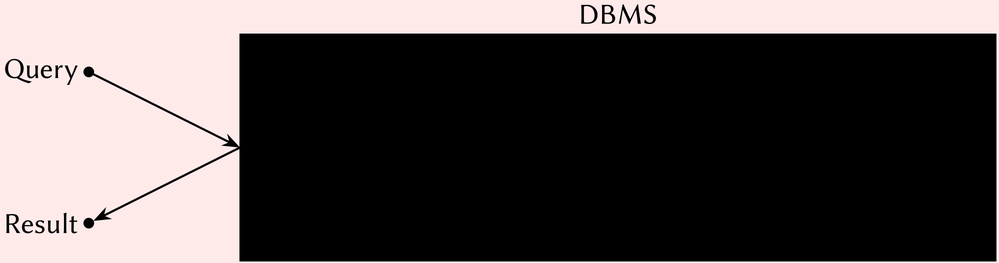
Hardware	CPUs, memory, storage, ….
Environment	Networking, “the cloud”, ….
Data	Volumes, computations, ….

…still the book covers the basic concepts accurately.

Overview of a database system



Overview of a database system

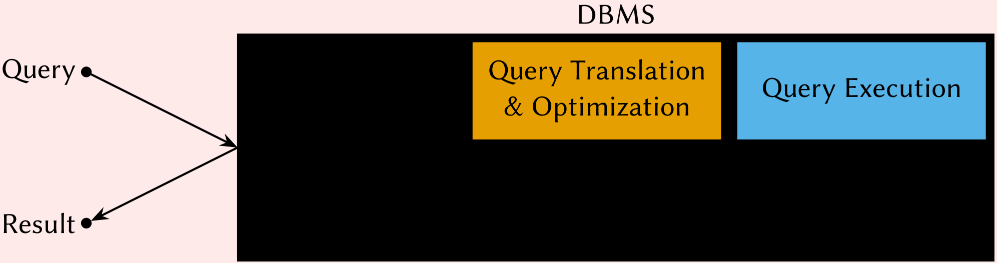


DBMS

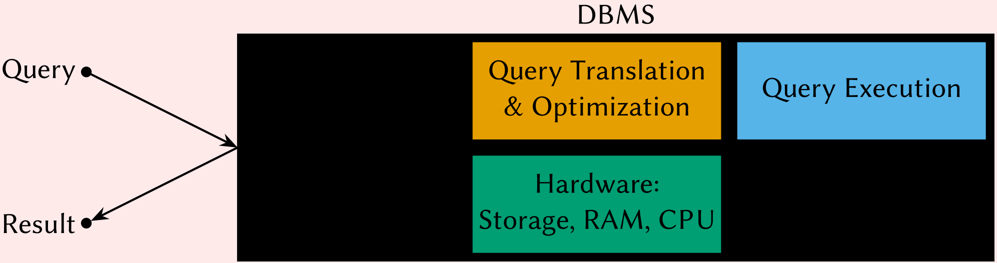
Query

Result

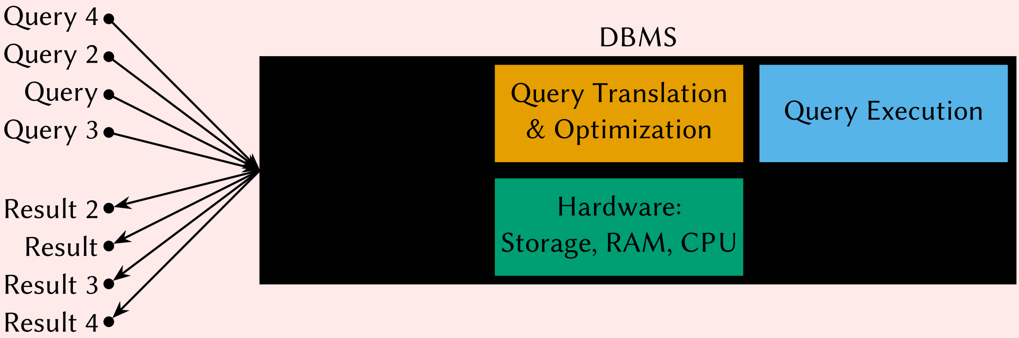
Overview of a database system



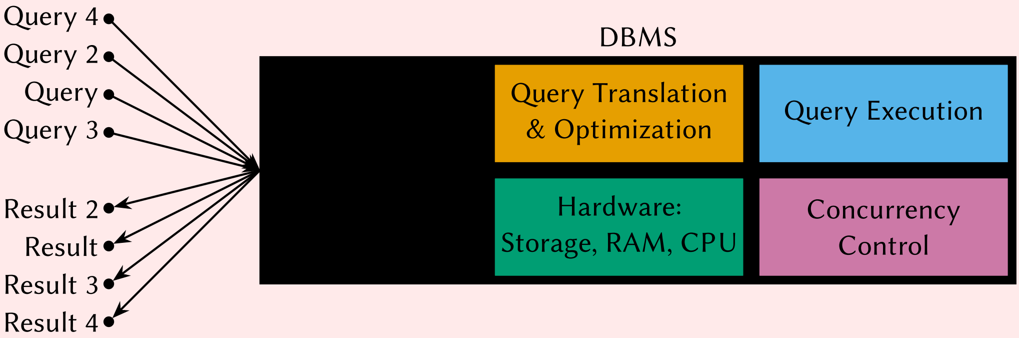
Overview of a database system



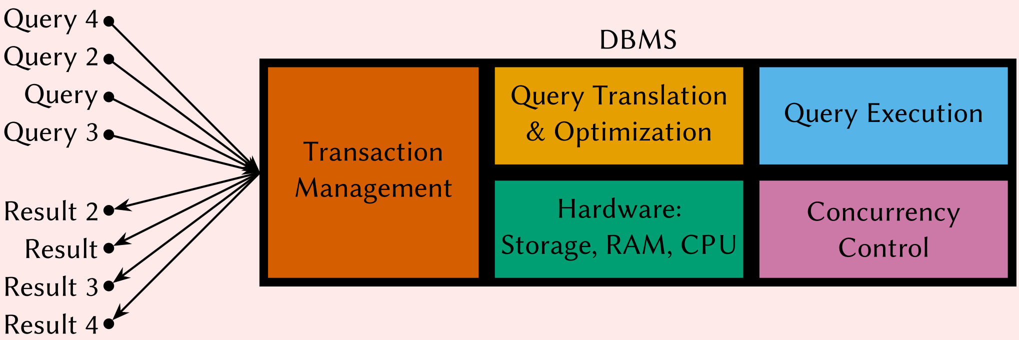
Overview of a database system



Overview of a database system



Overview of a database system



Hardware: Storage

A lot has changed, but basic principles are the same.

Hardware: Storage

A lot has changed, but basic principles are the same.

Storage Hierarchy

From *fast and expensive* to *slow and cheap*:

- CPU registers.
- CPU caches.
- Main Memory.
- Fast Storage (SSDs).
- Slow Storage (HDD).

Hardware: Storage

A lot has changed, but basic principles are the same.

Storage Hierarchy

From *fast and expensive* to *slow and cheap*:

- CPU registers.
- CPU caches.
- Main Memory.
- Fast Storage (SSDs).
- Slow Storage (HDD).

$\left.\begin{array}{l}\text{Main Memory.}\\\text{Fast Storage (SSDs).}\\\text{Slow Storage (HDD).}\end{array}\right\}$ Storage via network.

Hardware: Storage

A lot has changed, but basic principles are the same.

Storage Hierarchy

From *fast and expensive* to *slow and cheap*:

- CPU registers. ⟵ *out of our control.*
- CPU caches. ⟵ *out of our control (mostly).*
- Main Memory.
- Fast Storage (SSDs).
- Slow Storage (HDD).

} Storage via network.

Hardware: Storage in numbers

	Main Memory (DDR5 RAM)	Fast Storage (SSD, PCIe 4x)	Slow Storage (HDD, SATA-600)
Amount (similar price)	32GB	2TB	14TB
Speed (Read)	91 GiB/s	5.6 GiB/s	241 MiB/s
Speed (Write)	82 GiB/s	4.0 GiB/s	241 MiB/s
Latency (Read)	70 ns	96 µs	8.5 ms
Latency (Write)	70 ns	110 µs	9.5 ms

We used seek times for HDD as their latency.

Hardware: Storage in numbers

	Main Memory	Fast Storage	Slow Storage
	(DDR5 RAM)	(SSD, PCIe 4x)	(HDD, SATA-600)
Amount (similar price)	32GB	2TB	14TB
Speed (Read)	91 GiB/s	5.6 GiB/s	241 MiB/s
Speed (Write)	82 GiB/s	4.0 GiB/s	241 MiB/s
Latency (Read)	70 ns	96 µs	8.5 ms
Latency (Write)	70 ns	110 µs	9.5 ms
# Operations	⋙ 1 000 000 000/s	≈ 10 000/s	≈ 100/s

We used seek times for HDD as their latency.

Hardware: Storage in numbers

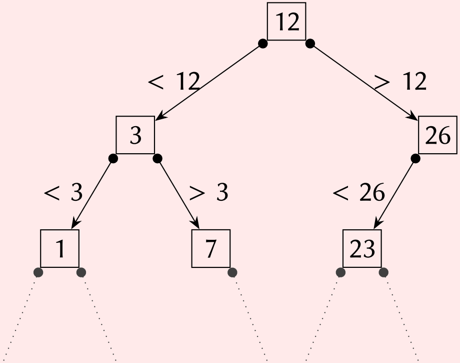
	Main Memory (DDR5 RAM)	Fast Storage (SSD, PCIe 4x)	Slow Storage (HDD, SATA-600)
Amount (similar price)	32GB	2TB	14TB
Speed (Read)	91 GiB/s	5.6 GiB/s	241 MiB/s
Speed (Write)	82 GiB/s	4.0 GiB/s	241 MiB/s
Latency (Read)	70 ns	96 µs	8.5 ms
Latency (Write)	70 ns	110 µs	9.5 ms
# Operations	$\ggg$ 1 000 000 000/s	≈ 10 000/s	≈ 100/s

Main memory *data structures and algorithms* are inefficient on permanent storage!

We used seek times for HDD as their latency.

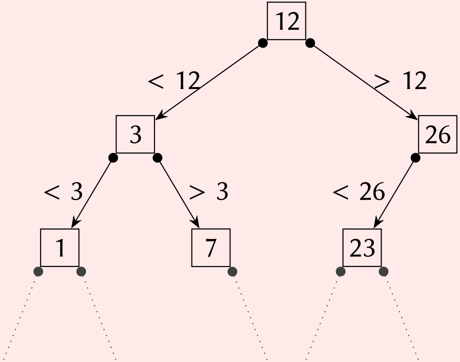
Example: Binary Search Trees vs B+ trees

Binary Search Trees: A *main memory* search structure



Example: Binary Search Trees vs B+ trees

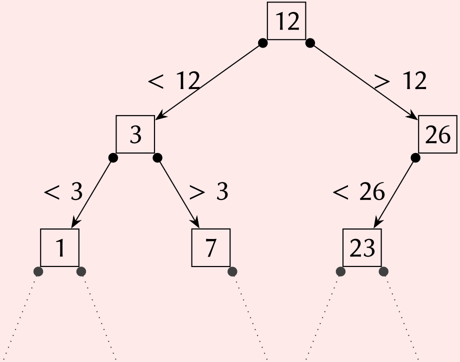
Binary Search Trees: A *main memory* search structure



- Each node holds a *search key* and value.
- Each node can point to up-to-two children.
- Leaves can have different *depths*.

Example: Binary Search Trees vs B+ trees

Binary Search Trees: A *main memory* search structure

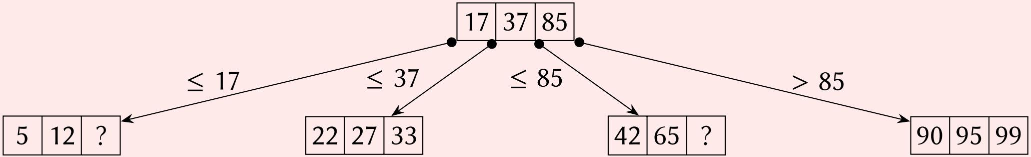


- Each node holds a *search key* and value.
- Each node can point to up-to-two children.
- Leaves can have different *depths*.

- Perfect balancing: $\leq \log_2(N)$ height.
- Realistic balancing: $\leq 2\log_2(N)$ height (e.g., red-black trees).
- SET in C++ and TREESET in Java.

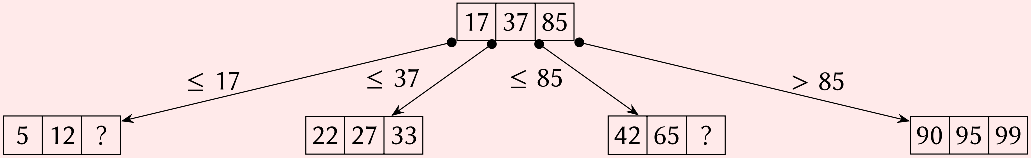
Example: Binary Search Trees vs B+ trees

B+ trees: An *external memory* search structure



Example: Binary Search Trees vs B+ trees

B+ trees: An *external memory* search structure

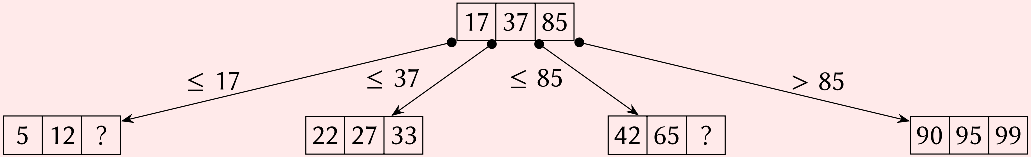


- Each node can hold K *search keys*.
- Each node can point to $K + 1$ children.
- Leaves all at the same *depth*.

Example: Binary Search Trees vs B+ trees

B+ trees: An *external memory* search structure

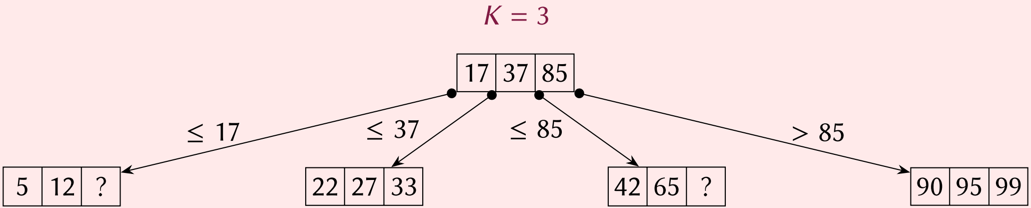
$K = 3$



- Each node can hold K *search keys*.
- Each node can point to $K + 1$ children.
- Leaves all at the same *depth*.

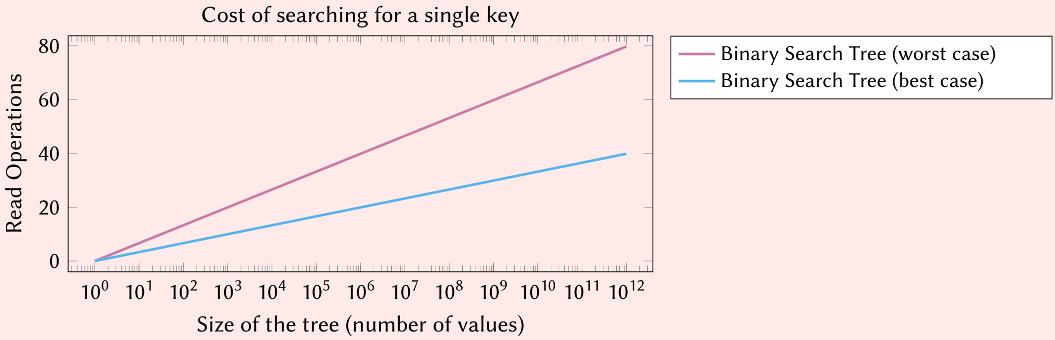
Example: Binary Search Trees vs B+ trees

B+ trees: An *external memory* search structure

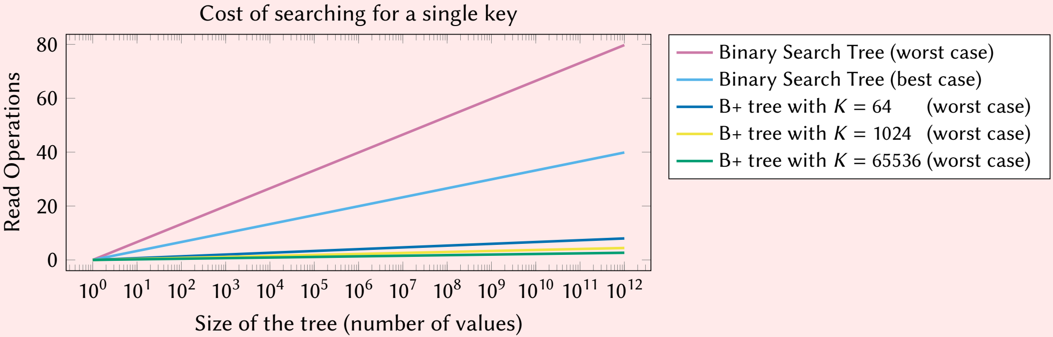


- Each node can hold K *search keys*.
- Each node can point to $K + 1$ children.
- Leaves all at the same *depth*.
- Nodes at-least half full.
- Height: $\leq \log_{K/2}(N)$ with N elements.

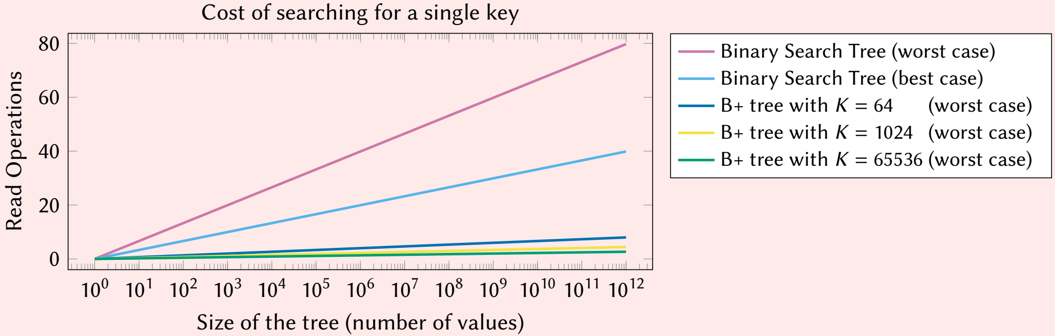
Example: Binary Search Trees vs B+ trees



Example: Binary Search Trees vs B+ trees



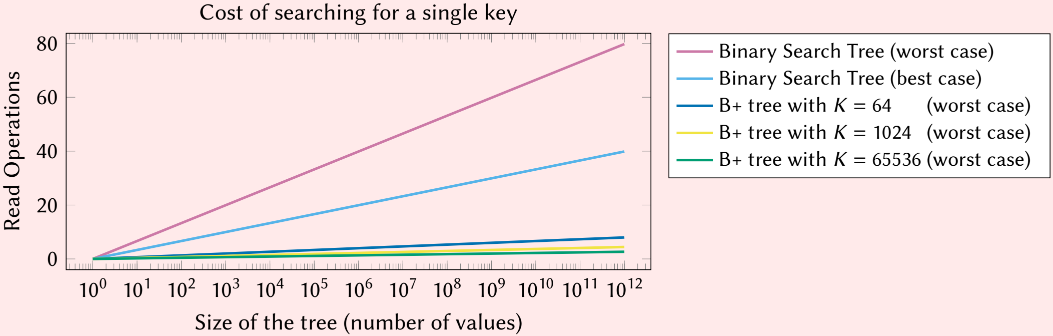
Example: Binary Search Trees vs B+ trees



Variants of B+ trees are the *standard* storage format for databases.

For example: Store relational tables using their *primary keys* as search keys.

Example: Binary Search Trees vs B+ trees



Variants of B+ trees are the *standard* storage format for databases.
For example: Store relational tables using their *primary keys* as search keys.

Other indices can use other index structures
(e.g., external memory variants of hash tables).

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Safe & easy approach

First: all operations of Query 1.
Then: all operations of Query 2.

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Safe & easy approach

First: all operations of Query 1.
Then: all operations of Query 2.

What if

- optimizing Query 2 takes a lot of time?
- the data of Query 2 is already in memory?
- the data of Query 2 is stored on another disk?

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Safe & easy approach

First: all operations of Query 1.
Then: all operations of Query 2.

What if

- optimizing Query 2 takes a lot of time?
- the data of Query 2 is already in memory?
- the data of Query 2 is stored on another disk?

For maximum performance: storage must *always* be busy!

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Performance requires concurrency

For maximum performance: storage must *always* be busy!

Perform operations in *parallel*.

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Performance requires concurrency

For maximum performance: storage must *always* be busy!

Perform operations in *parallel*.

This is *not just about* multi-threading.

Hardware: Concurrency

What if Query 1 needs to store data while Query 2 needs to read data?

Performance requires concurrency

For maximum performance: storage must *always* be busy!

Perform operations in *parallel*.

This is *not just about* multi-threading.

Say we have a single CPU core: we still want

- to perform IO *asynchronous*: no waiting on storage or network operations;
- to *overlap computations* of of one query with IO of others; and
- to *minimize storage operations* of queries (e.g., by combining them).

An example of concurrent execution

Consider a banking example in which

- Bo wants to transfer $400 to Alicia *if* Alicia has at-least $100 and
  Bo has at-least $700,
- Alicia wants to transfer $300 to Eve *if* Alicia has at-least $500,

and no account is allowed to have a negative balance.

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

Consider a banking example in which

- Bo wants to transfer \$400 to Alicia *if* Alicia has at-least \$100 and
  Bo has at-least \$700,
- Alicia wants to transfer \$300 to Eve *if* Alicia has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

$$\begin{array}{|c|l|} A & \$100 \\ B & \$300 \\ E & \$0 \end{array} \xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1} \begin{array}{|c|l|} A & \$500 \\ B & \$300 \\ E & \$0 \end{array}$$

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

$$\begin{array}{|c|l|}\hline A & \$100 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1} \begin{array}{|c|l|}\hline A & \$500 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{A \geq 500? \\ A := A - 400}]{\tau_2} \begin{array}{|c|l|}\hline A & \$200 \\ B & \$300 \\ E & \$0 \\ \hline\end{array}$$

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$$\xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1}$$

A	\$500
B	\$300
E	\$0

$$\xrightarrow[\substack{A \geq 500? \\ A := A - 400}]{\tau_2}$$

A	\$200
B	\$300
E	\$0

$$\xrightarrow[\substack{B \geq 700? \\ \text{(cancel)}}]{\tau_1}$$

A	\$200
B	\$300
E	\$0

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

$$\begin{array}{|c|c|} \hline A & \$100 \\ B & \$300 \\ E & \$0 \\ \hline \end{array} \xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1} \begin{array}{|c|c|} \hline A & \$500 \\ B & \$300 \\ E & \$0 \\ \hline \end{array} \xrightarrow[\substack{A \geq 500? \\ A := A - 400}]{\tau_2} \begin{array}{|c|c|} \hline A & \$200 \\ B & \$300 \\ E & \$0 \\ \hline \end{array} \xrightarrow[\substack{B \geq 700? \\ \text{(cancel)}}]{\tau_1}$$

$$\begin{array}{|c|c|} \hline A & \$200 \\ B & \$300 \\ E & \$0 \\ \hline \end{array} \xrightarrow[\substack{A := A - 400 \\ \text{(undo)}}]{\tau_1} \begin{array}{|c|c|} \hline A & \text{-}\$200 \\ B & \$300 \\ E & \$0 \\ \hline \end{array}$$

Source: https://doi.org/10.14778/3476249.3476275.

An example of concurrent execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

$$\begin{array}{|c|c|}\hline A & \$100 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1} \begin{array}{|c|c|}\hline A & \$500 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{A \geq 500? \\ A := A - 400}]{\tau_2} \begin{array}{|c|c|}\hline A & \$200 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{B \geq 700? \\ \text{(cancel)}}]{\tau_1}$$

$$\begin{array}{|c|c|}\hline A & \$200 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[\substack{A := A - 400 \\ \text{(undo)}}]{\tau_1} \begin{array}{|c|c|}\hline A & \text{-}\$200 \\ B & \$300 \\ E & \$0 \\ \hline\end{array} \xrightarrow[E := E + 300]{\tau_2} \begin{array}{|c|c|}\hline A & \text{-}\$200 \\ B & \$300 \\ E & \$300 \\ \hline\end{array}$$

Source: https://doi.org/10.14778/3476249.3476275.

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:

the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;
- a set of queries;

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;
- a set of queries;
- a interactive dialog between DBMS and program;
- ....

The ACID properties

Contract between a DBMS and its users.

The ACID properties

Contract between a DBMS and its users.

Transaction execution of transaction τ is

Atomic either all actions of τ are carried out (*commit*) or none are (*abort*);

Consistent transaction execution preserves the consistency of data;

Isolated τ is not affected by concurrently executing transactions;

Durable if the DBMS says τ is completed, then its effects are permanent.

The ACID properties

Contract between a DBMS and its users.

Transaction execution of transaction τ is

Atomic either all actions of τ are carried out (*commit*) or none are (*abort*);

Consistent transaction execution preserves the consistency of data;

Isolated τ is not affected by concurrently executing transactions;

Durable if the DBMS says τ is completed, then its effects are permanent.

Assumption: individual transactions *make sense* (do not violate consistency).

The ACID properties

Contract between a DBMS and its users.

Transaction execution of transaction τ is

Atomic either all actions of τ are carried out (*commit*) or none are (*abort*);

Consistent transaction execution preserves the consistency of data;

Isolated τ is not affected by concurrently executing transactions;

Durable if the DBMS says τ is completed, then its effects are permanent.

Assumption: individual transactions *make sense* (do not violate consistency).

Durability is *strong*: crashing or killing the DBMS program, power outage, ...

The ACID properties

Contract between a DBMS and its users.

Transaction execution of transaction τ is

Atomic either all actions of τ are carried out (*commit*) or none are (*abort*);

Consistent transaction execution preserves the consistency of data;

Isolated τ is not affected by concurrently executing transactions;

Durable if the DBMS says τ is completed, then its effects are permanent.

Assumption: individual transactions *make sense* (do not violate consistency).

Durability is *strong*: crashing or killing the DBMS program, power outage, ...
Typical assumption: *storage* is permanent & reliable.

An example of concurrent execution–revisited

Consider a banking example in which

- Bo wants to transfer $400 to Alicia *if* Alicia has at-least $100 and
  Bo has at-least $700,
- Alicia wants to transfer $300 to Eve *if* Alicia has at-least $500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An example of concurrent execution–revisited

Consider a banking example in which

- Bo wants to transfer \$400 to Alicia *if* Alicia has at-least \$100 and
  Bo has at-least \$700,
- Alicia wants to transfer \$300 to Eve *if* Alicia has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Transactions τ_1 and τ_2 make sense:
their isolated execution will never make balances negative.

An example of concurrent execution–revisited

Consider a banking example in which

- Bo wants to transfer \$400 to Alicia *if* Alicia has at-least \$100 and
  Bo has at-least \$700,
- Alicia wants to transfer \$300 to Eve *if* Alicia has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Transactions τ_1 and τ_2 make sense:

their isolated execution will never make balances negative.

Guarantee by an ACID-compliant database

No account will ever have a negative balance.

Serializability: A high standard for isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Serializability: A high standard for isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Serializability: A high standard for isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability: A high standard for isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability assumes *aborted* transactions have no side effects.

Serializability: A high standard for isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability assumes *aborted* transactions have no side effects.
This is not always the case (example later).

Simplified transaction notation

Consider the transaction τ:

τ = “if *Alicia* has \$500 and *Bo* has \$200, then
move \$400 from *Alicia* to *Eva*;
move \$100 from *Bo* to *Eva*”.

Simplified transaction notation

Consider the transaction τ:

τ = “if *Alicia* has \$500 and *Bo* has \$200, then
move \$400 from *Alicia* to *Eva*;
move \$100 from *Bo* to *Eva*”.

What are the operations of τ?

Depending on *how* the DBMS executes τ and the database state:

- Might read from *Alicia*’s account.
- Might read from *Bo*’s account.
- Might write to *Alicia*’s account.
- Might write to *Bo*’s account.
- Might write to *Eva*’s account.

Simplified transaction notation

Consider the transaction τ:

τ = “if *Alicia* has \$500 and *Bo* has \$200, then
move \$400 from *Alicia* to *Eva*;
move \$100 from *Bo* to *Eva*”.

Simplifying assumption

Each transaction is a sequence of read and write operations ending in *commit* or *abort*.

Simplified transaction notation

Consider the transaction τ:

$$\tau = \text{“if } \textit{Alicia} \text{ has \$500 and } \textit{Bo} \text{ has \$200, then}$$
$$\text{move \$400 from } \textit{Alicia} \text{ to } \textit{Eva};$$
$$\text{move \$100 from } \textit{Bo} \text{ to } \textit{Eva}\text{”.}$$

Simplifying assumption

Each transaction is a sequence of read and write operations ending in *commit* or *abort*.

$\text{Read}_\tau(\mathit{Alicia}), \text{Read}_\tau(\mathit{Bo}), \text{Write}_\tau(\mathit{Alicia}), \text{Write}_\tau(\mathit{Bo}), \text{Read}_\tau(\mathit{Eva}), \text{Write}_\tau(\mathit{Eva}), \text{Commit}_\tau.$

Simplified transaction notation

Consider the transaction τ:

τ = "if *Alicia* has \$500 and *Bo* has \$200, then
move \$400 from *Alicia* to *Eva*;
move \$100 from *Bo* to *Eva*".

Simplifying assumption

Each transaction is a sequence of read and write operations ending in *commit* or *abort*.

$\text{Read}_\tau(\mathit{Alicia}), \text{Read}_\tau(\mathit{Bo}), \text{Write}_\tau(\mathit{Alicia}), \text{Write}_\tau(\mathit{Bo}), \text{Read}_\tau(\mathit{Eva}), \text{Write}_\tau(\mathit{Eva}), \text{Commit}_\tau.$

The book writes $R_\tau(O)$ and $W_\tau(O)$ instead of $\text{Read}_\tau(O)$ and $\text{Write}_\tau(O)$.

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance (initial)	
A	\$100
B	\$300
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance
(initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}

Instance
(final)

A	\$100
B	\$300
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)	
A	\$100
B	\$800
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

Read$_{\tau_1}(A)$	
Write$_{\tau_1}(A)$	
Read$_{\tau_1}(B)$	
Write$_{\tau_1}(B)$	
Commit$_{\tau_1}$	
	Read$_{\tau_2}(A)$
	Write$_{\tau_2}(A)$
	Read$_{\tau_2}(E)$
	Write$_{\tau_2}(E)$
	Commit$_{\tau_2}$

Instance (final)

A	\$200
B	\$400
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance
(initial)

A	\$100
B	\$800
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

	Read$_{\tau_2}(A)$
	Abort$_{\tau_2}$
Read$_{\tau_1}(A)$	
Write$_{\tau_1}(A)$	
Read$_{\tau_1}(B)$	
Write$_{\tau_1}(B)$	
Commit$_{\tau_1}$	

Instance (final)

A	\$500
B	\$400
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Alicia has sufficient funds)

Instance (initial)	
A	\$500
B	\$300
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Alicia has sufficient funds)

Instance
(initial)

A	\$500
B	\$300
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Alicia has sufficient funds)

Instance (initial)

A	\$500
B	\$300
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

Instance (final)

A	\$200
B	\$300
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance
(initial)

A	\$100
B	\$300
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(B)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(B)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

Instance (final)

A	-\$200
B	\$300
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance
(initial)

A	\$500
B	\$800
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance
(initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

Instance
(final)

A	\$900
B	\$400
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance
(initial)

A	\$500
B	\$800
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	Read$_{\tau_2}(A)$
Read$_{\tau_1}(A)$	
Write$_{\tau_1}(A)$	
Read$_{\tau_1}(B)$	
Write$_{\tau_1}(B)$	
Commit$_{\tau_1}$	
	Write$_{\tau_2}(A)$
	Read$_{\tau_2}(E)$
	Write$_{\tau_2}(E)$
	Commit$_{\tau_2}$

Instance (final)

A	\$200
B	\$400
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	Read$_{\tau_2}(A)$
	Write$_{\tau_2}(A)$
Read$_{\tau_1}(A)$	
Write$_{\tau_1}(A)$	
	Read$_{\tau_2}(E)$
	Write$_{\tau_2}(E)$
Read$_{\tau_1}(B)$	
Write$_{\tau_1}(B)$	
	Commit$_{\tau_2}$
Commit$_{\tau_1}$	

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
	Commit_{τ_2}
Commit_{τ_1}	

Instance (final)

A	\$600
B	\$400
E	\$300

An example of schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Key observation: Serial schedules

Individual transactions *make sense* (do not violate consistency):

- No balance will ever get negative.
- No money disappears or appears out of thin air.

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the database*, executes, *releases the lock*.

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the database*, executes, *releases the lock*.

This will enforce a *serial schedule*.

Guaranteeing isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program (the DBMS).
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the database*, executes, *releases the lock*.

This will enforce a *serial schedule* and eliminate any concurrency.

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.

E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ….

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Using fine-grained locks

A transaction τ that wants to access database object O will:

- waits until it obtains a lock on O ($\text{Lock}_\tau(O)$),
- then perform its operations on O (e.g., $\text{Read}_\tau(O)$ and $\text{Write}_\tau(O)$), and
- finally release the lock on O ($\text{Release}_\tau(O)$).

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

Instance (final)

A	\$900
B	\$400
E	\$300

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance
(initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance
(initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ….

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues …

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	…
	Commit_{τ_2}

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	...
	Commit_{τ_2}
...	
Commit_{τ_1}	

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ….

In our examples we abstract from the details: *accounts* are database objects.

Lock-based access solves *some* issues …

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	…
	Commit_{τ_2}
…	
Commit_{τ_1}	

Instance (final)

A	\$600
B	\$400
E	\$300

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

...but not *all* issues ...

Instance (initial)

A	\$100
B	\$300
E	\$0

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

...but not *all* issues ...

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

Lock$_{\tau_1}(A)$	
Read$_{\tau_1}(A)$	
Write$_{\tau_1}(A)$	
Release$_{\tau_1}(A)$	
	Lock$_{\tau_2}(A)$
	Read$_{\tau_2}(A)$
	Write$_{\tau_2}(A)$
	...
	Commit$_{\tau_2}$
...	
Abort$_{\tau_1}$	

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ….

In our examples we abstract from the details: *accounts* are database objects.

…but not *all* issues …

Instance
(initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	…
	Commit_{τ_2}
…	
Abort_{τ_1}	

Instance
(final)

A	-\$200
B	\$300
E	\$300

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

…and introduces *new* issues.
Consider two transactions that both want to access *Alicia* and *Bo*:

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \dots; \qquad \tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_1}(A), \dots$$

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

…and introduces *new* issues.
Consider two transactions that both want to access *Alicia* and *Bo*:

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \ldots; \qquad \tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_1}(A), \ldots$$

Schedule

$\mathsf{Lock}_{\tau_1}(A)$	
	$\mathsf{Lock}_{\tau_2}(B)$
$\mathsf{Lock}_{\tau_1}(B)$	
	$\mathsf{Lock}_{\tau_2}(A)$

Improving isolation using locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., tables, rows, blocks of memory, blocks in on-disk data structures, ...

In our examples we abstract from the details: *accounts* are database objects.

…and introduces *new* issues.
Consider two transactions that both want to access *Alicia* and *Bo*:

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \dots; \qquad \tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_1}(A), \dots$$

Schedule

$\mathsf{Lock}_{\tau_1}(A)$	
	$\mathsf{Lock}_{\tau_2}(B)$
$\mathsf{Lock}_{\tau_1}(B)$	
	$\mathsf{Lock}_{\tau_2}(A)$

Both transactions will wait forever: a deadlock!

Achieving serializability with locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) do guarantee *serializability*.

Achieving serializability with locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_{\tau}(O)$.

Achieving serializability with locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_\tau(O)$.

Strict 2PL: locks are only released after completion (Commit_τ or Abort_τ).

Achieving serializability with locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_{\tau}(O)$.

Strict 2PL: locks are only released after completion (Commit_{τ} or Abort_{τ}).

Notice—Nothing to deal with *deadlocks*.

An example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathsf{Lock}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),\\ & \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);\\ \tau_2 = {} & \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),\\ & \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).\end{aligned}$$

An example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),\\ & \quad \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);\\ \tau_2 = {} & \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),\\ & \quad \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).\end{aligned}$$

An example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(A), \text{Read}_{\tau_1}(B), \text{Write}_{\tau_1}(B),$$
$$\text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(B), \text{Release}_{\tau_1}(A);$$
$$\tau_2 = \text{Lock}_{\tau_2}(E), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(A), \text{Write}_{\tau_2}(A), \text{Read}_{\tau_2}(E), \text{Write}_{\tau_2}(E),$$
$$\text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(E).$$

An example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700? B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B), \\ & \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(B), \mathsf{Release}_{\tau_1}(A); \\ \tau_2 = {} & \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E), \\ & \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).\end{aligned}$$

These are all *strict* 2PL: locks are released after the transactions commit.

An example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B), \\ & \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(B), \mathsf{Release}_{\tau_1}(A); \\ \tau_2 = {} & \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E), \\ & \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).\end{aligned}$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

An example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(A), \text{Read}_{\tau_1}(B), \text{Write}_{\tau_1}(B),\\ & \quad \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(B), \text{Release}_{\tau_1}(A);\\ \tau_2 = {} & \text{Lock}_{\tau_2}(E), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(A), \text{Write}_{\tau_2}(A), \text{Read}_{\tau_2}(E), \text{Write}_{\tau_2}(E),\\ & \quad \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(E).\end{aligned}$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

- If τ_1 executes $\text{Lock}_{\tau_1}(A)$ *before* τ_2 executes $\text{Lock}_{\tau_2}(A)$:
  all read and write operations of τ_1 effectively happen before those of τ_2.

An example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),\\ & \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(B), \mathsf{Release}_{\tau_1}(A);\\ \tau_2 = {} & \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),\\ & \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).\end{aligned}$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

- If τ_1 executes $\mathsf{Lock}_{\tau_1}(A)$ *before* τ_2 executes $\mathsf{Lock}_{\tau_2}(A)$:
  all read and write operations of τ_1 effectively happen before those of τ_2.
- If τ_2 executes $\mathsf{Lock}_{\tau_2}(A)$ *before* τ_1 executes $\mathsf{Lock}_{\tau_1}(A)$:
  all read and write operations of τ_2 effectively happen before those of τ_1.

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \ldots, \tau_n$ is identical to some serial schedule.

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \ldots, \tau_n$ is identical to some serial schedule.

We are going to build a graph G in which:

- every transaction is a node;
- there is an edge (τ, τ') if τ obtained its final lock in Σ *before* τ'.

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \ldots, \tau_n$ is identical to some serial schedule.

We are going to build a graph G in which:

- every transaction is a node;
- there is an edge (τ, τ') if τ obtained its final lock in Σ *before* τ'.

Claim: Graph G is a directed acyclic graph

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \ldots, \tau_n$ is identical to some serial schedule.

We are going to build a graph G in which:

- every transaction is a node;
- there is an edge (τ, τ') if τ obtained its final lock in Σ *before* τ'.

Claim: Graph G is a directed acyclic graph

By contradiction: what would a cycle involving distinct transactions τ and τ' represent?

Two-phase locking provides serializability (sketch)

Let $\tau_1, \dots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \dots, \tau_n$ is identical to some serial schedule.

We are going to build a graph G in which:

- every transaction is a node;
- there is an edge (τ, τ') if τ obtained its final lock in Σ *before* τ'.

Claim: Graph G is a directed acyclic graph

By contradiction: what would a cycle involving distinct transactions τ and τ' represent?

Claim: Any topological ordering of G is a *serial schedule* that is identical to Σ

Two-phase locking provides serializability (sketch)

Let $\tau_1, \ldots, \tau_n$ be a set of (committed) transactions executed using 2PL.

Prove: any 2PL-adhering schedule Σ for $\tau_1, \ldots, \tau_n$ is identical to some serial schedule.

We are going to build a graph G in which:

- every transaction is a node;
- there is an edge (τ, τ') if τ obtained its final lock in Σ *before* τ'.

Claim: Graph G is a directed acyclic graph

By contradiction: what would a cycle involving distinct transactions τ and τ' represent?

Claim: Any topological ordering of G is a *serial schedule* that is identical to Σ

Idea: all operations of τ can be moved until right after it obtained all its locks,
but before the next transaction obtained all its locks.

Two-phase locking and deadlocks

Consider the transactions

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(B), \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);$$
$$\tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(B), \mathsf{Write}_{\tau_2}(A), \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(B).$$

These transactions are strict 2PL.

Two-phase locking and deadlocks

Consider the transactions

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(B), \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);$$

$$\tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(B), \mathsf{Write}_{\tau_2}(A), \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(B).$$

These transactions are strict 2PL.

Some schedules will cause a deadlock

Schedule

$\mathsf{Lock}_{\tau_1}(A)$	
	$\mathsf{Lock}_{\tau_2}(B)$
$\mathsf{Lock}_{\tau_1}(B)$	
	$\mathsf{Lock}_{\tau_2}(A)$

Two-phase locking and deadlocks

Consider the transactions

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(A), \text{Release}_{\tau_1}(B);$$
$$\tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(B), \text{Write}_{\tau_2}(A), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(B).$$

These transactions are strict 2PL.

Some schedules will cause a deadlock

Schedule

$\text{Lock}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(B)$
$\text{Lock}_{\tau_1}(B)$	
	$\text{Lock}_{\tau_2}(A)$

Deadlocks are one of the issues arising from *lock contention*.

Dealing with deadlocks: Pessimistic approach

Pessimistic: make sure deadlocks *cannot happen*

Dealing with deadlocks: Pessimistic approach

Pessimistic: make sure deadlocks *cannot happen*

Enforce that all transactions obtain their locks in a unique predetermined order.
E.g., first locks on Alicia, then Bo, then Celeste, then Dafni, then Eva, ...

Dealing with deadlocks: Pessimistic approach

Pessimistic: make sure deadlocks *cannot happen*

Enforce that all transactions obtain their locks in a unique predetermined order.
E.g., first locks on Alicia, then Bo, then Celeste, then Dafni, then Eva, ….

Need to determine all database objects to lock up-front: often an over-approximation!

Dealing with deadlocks: Pessimistic approach

Pessimistic: make sure deadlocks *cannot happen*

Enforce that all transactions obtain their locks in a unique predetermined order.
E.g., first locks on Alicia, then Bo, then Celeste, then Dafni, then Eva, ...

Need to determine all database objects to lock up-front: often an over-approximation!

Example

Consider the transaction

$$\tau = \text{“if } \textit{Bo} \text{ has \$500, then move \$200 from } \textit{Bo} \text{ to } \textit{Alicia}\text{”.}$$

Any schedule for τ needs to start with:

$$\mathsf{Lock}_{\tau}(\mathit{Alicia}), \mathsf{Lock}_{\tau}(\mathit{Bo}), \ldots,$$

we even lock Alicia if Bo does *not have funds*.

Dealing with deadlocks: Optimistic approach

Optimistic: detect deadlocks and *deal with them*

- Detect lack of progress of certain transactions due to deadlocks.
  E.g., using timeouts, which can result in *false positives*.

Dealing with deadlocks: Optimistic approach

Optimistic: detect deadlocks and *deal with them*

- Detect lack of progress of certain transactions due to deadlocks.
  E.g., using timeouts, which can result in *false positives*.

- On detection of a deadlock: Restart involved transactions.
  Requires the ability to *rollback and restart* transactions.

Dealing with deadlocks: Optimistic approach

Optimistic: detect deadlocks and *deal with them*

- Detect lack of progress of certain transactions due to deadlocks.
  E.g., using timeouts, which can result in *false positives*.

- On detection of a deadlock: Restart involved transactions.
  Requires the ability to *rollback and restart* transactions.

- Very flexible in how transactions obtain locks.
  E.g., can minimize the number of locks used and usage duration.

Dealing with deadlocks: Optimistic approach

Optimistic: detect deadlocks and *deal with them*

- Detect lack of progress of certain transactions due to deadlocks.
  E.g., using timeouts, which can result in *false positives*.

- On detection of a deadlock: Restart involved transactions.
  Requires the ability to *rollback and restart* transactions.

- Very flexible in how transactions obtain locks.
  E.g., can minimize the number of locks used and usage duration.

Typically used in systems that employ locks.

Dealing with deadlocks: Super-Optimistic approach

Super-Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

Dealing with deadlocks: Super-Optimistic approach

Super-Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for deadlock detection: e.g., no need for reliable timeouts.

Dealing with deadlocks: Super-Optimistic approach

Super-Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for deadlock detection: e.g., no need for reliable timeouts.
- Very easy to implement.

Dealing with deadlocks: Super-Optimistic approach

Super-Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for deadlock detection: e.g., no need for reliable timeouts.
- Very easy to implement.
- Minimizes the costs for transactions that are able to commit.

Dealing with deadlocks: Super-Optimistic approach

Super-Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for deadlock detection: e.g., no need for reliable timeouts.
- Very easy to implement.
- Minimizes the costs for transactions that are able to commit.
- Will perform badly when there is a high amount of lock-contention.

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Consider the following incomplete schedule:

$\text{Lock}_{\tau_1}(A)$		
$\text{Read}_{\tau_1}(A)$		
$\text{Write}_{\tau_1}(A)$		
$\text{Release}_{\tau_1}(A)$		
	$\text{Lock}_{\tau_2}(A)$	
	$\text{Read}_{\tau_2}(A)$	
	$\text{Write}_{\tau_2}(A)$	
	Commit_{τ_2}	
	$\text{Release}_{\tau_2}(A)$	
		$\text{Lock}_{\tau_3}(A)$
		$\text{Read}_{\tau_3}(A)$

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Consider the following incomplete schedule:

$\text{Lock}_{\tau_1}(A)$		
$\text{Read}_{\tau_1}(A)$		
$\text{Write}_{\tau_1}(A)$		
$\text{Release}_{\tau_1}(A)$		
	$\text{Lock}_{\tau_2}(A)$	
	$\text{Read}_{\tau_2}(A)$	
	$\text{Write}_{\tau_2}(A)$	
	Commit_{τ_2}	
	$\text{Release}_{\tau_2}(A)$	
		$\text{Lock}_{\tau_3}(A)$
		$\text{Read}_{\tau_3}(A)$

τ_1 gets aborted.

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Consider the following incomplete schedule:

$\mathsf{Lock}_{\tau_1}(A)$		
$\mathsf{Read}_{\tau_1}(A)$		
$\mathsf{Write}_{\tau_1}(A)$		
$\mathsf{Release}_{\tau_1}(A)$		
	$\mathsf{Lock}_{\tau_2}(A)$	
	$\mathsf{Read}_{\tau_2}(A)$	
	$\mathsf{Write}_{\tau_2}(A)$	
	Commit_{τ_2}	
	$\mathsf{Release}_{\tau_2}(A)$	
		$\mathsf{Lock}_{\tau_3}(A)$
		$\mathsf{Read}_{\tau_3}(A)$

τ_1 gets aborted.

Can we rollback $\mathsf{Write}_{\tau_1}(A)$?

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Consider the following incomplete schedule:

$\text{Lock}_{\tau_1}(A)$		
$\text{Read}_{\tau_1}(A)$		
$\text{Write}_{\tau_1}(A)$		
$\text{Release}_{\tau_1}(A)$		
	$\text{Lock}_{\tau_2}(A)$	
	$\text{Read}_{\tau_2}(A)$	
	$\text{Write}_{\tau_2}(A)$	
	Commit_{τ_2}	
	$\text{Release}_{\tau_2}(A)$	
		$\text{Lock}_{\tau_3}(A)$
		$\text{Read}_{\tau_3}(A)$

τ_1 gets aborted.

How do we rollback Commit_{τ_2}?

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Consider the following incomplete schedule:

$\text{Lock}_{\tau_1}(A)$		
$\text{Read}_{\tau_1}(A)$		
$\text{Write}_{\tau_1}(A)$		
$\text{Release}_{\tau_1}(A)$		
	$\text{Lock}_{\tau_2}(A)$	
	$\text{Read}_{\tau_2}(A)$	
	$\text{Write}_{\tau_2}(A)$	
	Commit_{τ_2}	
	$\text{Release}_{\tau_2}(A)$	
		$\text{Lock}_{\tau_3}(A)$
		$\text{Read}_{\tau_3}(A)$

τ_1 gets aborted.

Should we also rollback $\text{Read}_{\tau_3}(A)$

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Problem

2PL can result in an *unrecoverable* schedule with *cascading aborts*.

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Problem

2PL can result in an *unrecoverable* schedule with *cascading aborts*.
Cause: Future transactions can read uncommitted changes.

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Problem

2PL can result in an *unrecoverable* schedule with *cascading aborts*.
Cause: Future transactions can read uncommitted changes.

Simple fix: use *Strict* 2PL.
In *Strict* 2PL: locks are only released after completion (Commit$_\tau$ or Abort$_\tau$).

Two-phase locking and rollbacks

Serializability is about *committed* transactions!

Problem

2PL can result in an *unrecoverable* schedule with *cascading aborts*.
Cause: Future transactions can read uncommitted changes.

Simple fix: use *Strict* 2PL.
In *Strict* 2PL: locks are only released after completion (Commit$_\tau$ or Abort$_\tau$).

Future transactions can only read the outcome of this transaction *after* it releases the lock: this is always after any changes have been committed—no *uncommitted* reads.

The cost of locks

- Locks have *overheads*:
  e.g., storing locks, meta-data to manage waiting and deadlock detection, …

The cost of locks

- Locks have *overheads*:
  e.g., storing locks, meta-data to manage waiting and deadlock detection, …

- Locks limit *concurrency*:
  no two transactions can work on the same data at the same time.

The cost of locks

- Locks have *overheads*:
  e.g., storing locks, meta-data to manage waiting and deadlock detection, …
- Locks limit *concurrency*:
  no two transactions can work on the same data at the same time.
- Locks increase *latencies*:
  obtaining locks takes time, especially when locks are already in use.

The cost of locks

- Locks have *overheads*:
  e.g., storing locks, meta-data to manage waiting and deadlock detection, …
- Locks limit *concurrency*:
  no two transactions can work on the same data at the same time.
- Locks increase *latencies*:
  obtaining locks takes time, especially when locks are already in use.

Example

Consider transactions τ_1 and τ_2 such that:

$$\tau_1 \text{ writes to data items } O_1, \dots, O_{10}, \qquad \tau_2 \text{ only writes to data item } O_1.$$

If τ_1 obtains the lock on O_1 first: τ_2 has to *wait* until τ_1 finishes!

The cost of locks

- Locks have *overheads*:
  e.g., storing locks, meta-data to manage waiting and deadlock detection, …
- Locks limit *concurrency*:
  no two transactions can work on the same data at the same time.
- Locks increase *latencies*:
  obtaining locks takes time, especially when locks are already in use.

Example

Consider transactions τ_1 and τ_2 such that:

τ_1 writes to data items $O_1, \dots, O_{10}$, τ_2 only writes to data item O_1.

If τ_1 obtains the lock on O_1 first: τ_2 has to *wait* until τ_1 finishes!

What if some long-running transaction τ_0 holds the lock on O_{10}?

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Consider transactions τ_1 and τ_2 operating on data object O.

- Case 1: τ_1 and τ_2 both only *read* O:
  The transactions do not affect each other: This is *fine*.

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Consider transactions τ_1 and τ_2 operating on data object O.

- Case 1: τ_1 and τ_2 both only *read* O:
  The transactions do not affect each other: This is *fine*.

- Case 2: τ_1 *writes* O after which τ_2 *reads* O:
  A *dirty read*: τ_2 can read *uncommitted* data of τ_1.

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Consider transactions τ_1 and τ_2 operating on data object O.

- Case 1: τ_1 and τ_2 both only *read* O:
  The transactions do not affect each other: This is *fine*.
- Case 2: τ_1 *writes* O after which τ_2 *reads* O:
  A *dirty read*: τ_2 can read *uncommitted* data of τ_1.
- Case 3: τ_1 *reads* O after which τ_2 *writes* O:
  A *unrepeatable read*: if τ_1 reads O again, it might read different values!

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Consider transactions τ_1 and τ_2 operating on data object O.

- Case 1: τ_1 and τ_2 both only *read* O:
  The transactions do not affect each other: This is *fine*.
- Case 2: τ_1 *writes* O after which τ_2 *reads* O:
  A *dirty read*: τ_2 can read *uncommitted* data of τ_1.
- Case 3: τ_1 *reads* O after which τ_2 *writes* O:
  A *unrepeatable read*: if τ_1 reads O again, it might read different values!
- Case 4: τ_1 and τ_2 both *write* O:
  A *lost update*: τ_2 can overwrite updates made by τ_1.

Issues resolved by using locks

Locks deal with *concurrency*: two transactions operating on the same data.

Consider transactions τ_1 and τ_2 operating on data object O.

- Case 1: τ_1 and τ_2 both only *read* O:
  The transactions do not affect each other: This is *fine*.
- Case 2: τ_1 *writes* O after which τ_2 *reads* O (write-read conflict):
  A *dirty read*: τ_2 can read *uncommitted* data of τ_1.
- Case 3: τ_1 *reads* O after which τ_2 *writes* O (read-write conflict):
  A *unrepeatable read*: if τ_1 reads O again, it might read different values!
- Case 4: τ_1 and τ_2 both *write* O (write-write conflict):
  A *lost update*: τ_2 can overwrite updates made by τ_1.

We have seen earlier examples of these conflicts!

Further concurrency issues

Abstractions can hide issues!

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

τ_1 = "Recompute the age of all people that have their birth date."

τ_2 = "Add a new user Celeste that also has their birth date."

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

τ_1 = "Recompute the age of all people that have their birth date."

τ_2 = "Add a new user Celeste that also has their birth date."

First, τ_1 obtains locks on all *people* that have their birth date "tomorrow" (fine-grained).

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

τ_1 = "Recompute the age of all people that have their birth date."

τ_2 = "Add a new user Celeste that also has their birth date."

First, τ_1 obtains locks on all *people* that have their birth date "tomorrow" (fine-grained).
Then: τ_2 adds the new user Celeste, unbeknownst to τ_1.

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

$$\tau_1 = \text{“Recompute the age of all people that have their birth date.”}$$
$$\tau_2 = \text{“Add a new user Celeste that also has their birth date.”}$$

First, τ_1 obtains locks on all *people* that have their birth date “tomorrow” (fine-grained).
Then: τ_2 adds the new user Celeste, unbeknownst to τ_1.
Then: τ_1 updates all ages, and misses Celeste completely.

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

τ_1 = "Recompute the age of all people that have their birth date."

τ_2 = "Add a new user Celeste that also has their birth date."

First, τ_1 obtains locks on all *people* that have their birth date "tomorrow" (fine-grained).
Then: τ_2 adds the new user Celeste, unbeknownst to τ_1.
Then: τ_1 updates all ages, and misses Celeste completely.

This is called the *phantom* problem:
τ_1 needed a lock on *all possible rows*, but our abstraction does not have this type of lock!

Further concurrency issues

Abstractions can hide issues!

Our abstraction: A given set of database objects, reads and writes on individual objects.

What if database objects get added?

At the middle of the night:

τ_1 = "Recompute the age of all people that have their birth date."

τ_2 = "Add a new user Celeste that also has their birth date."

Solution for our abstraction: "predicate" locks

Accessing O: not just lock O, but also all *predicates* that include O:
τ_1 and τ_2 should both get a lock on predicate "birth date is tomorrow".

We won't look at how to implement this.

Practice: Read and write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Practice: Read and write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Practice: Read and write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Introduce separate read and write locks

- Multiple transactions can hold a lock at the same time *if they all hold read locks*.
- Only one transaction can hold a lock *if that transaction holds a write lock*.

Practice: Read and write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Introduce separate read and write locks

- Multiple transactions can hold a lock at the same time *if they all hold read locks*.
- Only one transaction can hold a lock *if that transaction holds a write lock*.

Result

- Many transactions can read at the same time.
- Read-write, write-read, and write-write conflicts are prevented.

The cost of serializability

- Serializability provides *strong* isolation guarantees.
- Providing these guarantees *will* impact concurrency
  (independent of the implementation mechanism, e.g., locks).

The cost of serializability

- Serializability provides *strong* isolation guarantees.
- Providing these guarantees *will* impact concurrency (independent of the implementation mechanism, e.g., locks).

Beyond locks

- Timestamp-based methods to detect and resolve conflicts.
- Processing transactions concurrently in batches.
- …

Each come with their own strengths and weaknesses.

The cost of serializability

- Serializability provides *strong* isolation guarantees.
- Providing these guarantees *will* impact concurrency (independent of the implementation mechanism, e.g., locks).

Beyond locks

- Timestamp-based methods to detect and resolve conflicts.
- Processing transactions concurrently in batches.
- …

Each come with their own strengths and weaknesses.

To improve performance, you can *give up* on serializability!

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **READ UNCOMMITTED**

- no read locks,
- *long-duration* write (and predicate) locks before writing data.

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **READ COMMITTED**

- *short-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

There are excellent papers on this topic! E.g., `https://doi.org/10.1145/568271.223785` and `https://doi.org/10.1016/0950-5849(96)01109-3` are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **REPEATABLE READ**

- *short-duration* predicate locks and *long-duration* read locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **SERIALIZABLE**

- *long-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of isolation in SQL

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **SERIALIZABLE** (2PL)

- *long-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Transactions in SQL

Start of a transaction: DBMS specific.

- Some want **START TRANSACTION**.
- Some run each query as a separate committed transaction by default.
- Some treat a sequence of queries as a single transaction by default.

Transactions in SQL

Start of a transaction: DBMS specific.

- Some want **START TRANSACTION**.
- Some run each query as a separate committed transaction by default.
- Some treat a sequence of queries as a single transaction by default.

COMMIT to end and commit a transaction.

Transactions in SQL

Start of a transaction: DBMS specific.

- Some want **START TRANSACTION**.
- Some run each query as a separate committed transaction by default.
- Some treat a sequence of queries as a single transaction by default.

COMMIT to end and commit a transaction.

ROLLBACK to abort the transaction.

Transactions in SQL

Start of a transaction: DBMS specific.

- Some want **START TRANSACTION**.
- Some run each query as a separate committed transaction by default.
- Some treat a sequence of queries as a single transaction by default.

COMMIT to end and commit a transaction.

ROLLBACK to abort the transaction.

SET TRANSACTION ISOLATION LEVEL x to set the isolation level x, one of:

- **SERIALIZABLE**,
- **REPEATABLE READ**,
- **READ COMMITTED**, or
- **READ UNCOMMITTED**.

Transactions in SQL

Start of a transaction: DBMS specific.

- Some want **START TRANSACTION**.
- Some run each query as a separate committed transaction by default.
- Some treat a sequence of queries as a single transaction by default.

COMMIT to end and commit a transaction.

ROLLBACK to abort the transaction.

SET TRANSACTION ISOLATION LEVEL x to set the isolation level x, one of:

- **SERIALIZABLE**,
- **REPEATABLE READ**,
- **READ COMMITTED**, or
- **READ UNCOMMITTED**.

SET TRANSACTION c to set whether the transaction can modify data, with c one of **READ WRITE** or **READ ONLY**.

Transactions in SQL

Example in DB2 (command line)

By default, each query in DB2 is a separate transaction (autocommit).
We start the DB2 command line with db +c to disable autocommit.

Transactions in SQL

Example in DB2 (command line)

By default, each query in DB2 is a separate transaction (autocommit).
We start the DB2 command line with db +c to disable autocommit.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION READ WRITE;
INSERT ...
UPDATE ...
COMMIT; -- or ROLLBACK to undo changes.
```

Transactions in SQL

Example in DB2 (command line)

By default, each query in DB2 is a separate transaction (autocommit).
We start the DB2 command line with db +c to disable autocommit.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION READ WRITE; -- Not a great idea!
INSERT ...
UPDATE ...
COMMIT; -- or ROLLBACK to undo changes.
```

Transactions in SQL

Example in DB2 (command line)

By default, each query in DB2 is a separate transaction (autocommit).
We start the DB2 command line with db +c to disable autocommit.

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION READ WRITE; -- Not a great idea!
INSERT …
UPDATE …
COMMIT; -- or **ROLLBACK** to undo changes.

Using savepoints, one can undo *parts* of transactions.