Fault-Tolerant Distributed Transactions on Blockchain
Toward Scalable Blockchain

Suyash Gupta  Jelle Hellings  Mohammad Sadoghi
Scalability versus Fully-Replicated Blockchains

*Scalability: adding resources $\rightarrow$ adding performance.*
Scalability versus Fully-Replicated Blockchains

*Scalability: adding resources $\Rightarrow$ adding performance.*

Full replication: adding resources (replicas) $\Rightarrow$ less performance!
Partition the system: More storage and potentially more performance. Potentially lower latencies if data ends up closer to users.
Partition the system: More storage and potentially more performance. Potentially lower latencies if data ends up closer to users.

Adding shards \(\Rightarrow\) adding throughput (parallel processing), adding storage.
Distributed Systems: Specialization

Specialize the system: Different nodes have distinct tasks.
Specialized hardware and software per task.
Distributed Systems: Specialization

Specialize the system: Different nodes have distinct tasks.
Specialized hardware and software *per* task.

Specializing roles $\rightarrow$ adding throughput (parallel processing, specialized hardware, ...).
Central Ideas for Improvement

Reminder
We can make a resilient system that manages data: e.g., fully-replicated blockchains.

▶ **Role Specialization**: make the storage system a blockchain.
   Requires: *reliable read-only updates of the blockchain.*
   Permissionless blockchains: light clients!

▶ **Sharding**: make each shard an independent blockchain.
   Requires: *reliable communication between blockchains.*
   Permissionless blockchains: relays, atomic swaps!
Central Ideas for Improvement

Reminder
We can make a resilient system that manages data: e.g., fully-replicated blockchains.

- **Role Specialization**: make the storage system a blockchain.
  Requires: *reliable read-only updates of the blockchain.*
  Permissionless blockchains: light clients!

- **Sharding**: make each shard an independent blockchain.
  Requires: *reliable communication between blockchains.*
  Permissionless blockchains: relays, atomic swaps!

Consensus is of no use here if we want efficiency.
Definition
Let $\mathcal{C}$ be a cluster deciding on a sequence of transactions $\mathcal{L}$ and $\mathcal{L}$ be a learner.

The **Byzantine learning problem** is the problem of sending $\mathcal{L}$ from $\mathcal{C}$ to $\mathcal{L}$ such that:

- the learner $\mathcal{L}$ will eventually *receive all* decided transactions;
- the learner $\mathcal{L}$ will *only receive* decided transactions.
Definition
Let $C$ be a cluster deciding on a sequence of transactions $\mathcal{L}$ and $L$ be a learner.

The **Byzantine learning problem** is the problem of sending $\mathcal{L}$ from $C$ to $L$ such that:

- the learner $L$ will eventually receive all decided transactions;
- the learner $L$ will only receive decided transactions.

Practical requirements

- Minimizing overall communication.
- Load balancing among all replicas in $C$. 
Definition
Let $v$ be a value with storage size $s = \|v\|$. An information dispersal algorithm can encode $v$ in $n$ pieces $v'$ such that $v$ can be decoded from every set of $n - f$ such pieces.

Theorem (Rabin 1989)
The IDA algorithm is an optimal information dispersal algorithm:
- Each piece $v'$ has size $\left\lfloor \frac{\|v\|}{n-f} \right\rfloor$.
- The $n - f$ pieces necessary for decoding have a total size of $(n - f) \left\lfloor \frac{\|v\|}{(n-f)} \right\rfloor \approx \|v\|$.
The Delayed-Replication Algorithm

Idea: $C$ sends a ledger to learner $L$
The Delayed-Replication Algorithm

Idea: $C$ sends a ledger to learner $L$

1. Partition the ledger in sequences $S$ of $n$ transactions.
The Delayed-Replication Algorithm

Idea: $C$ sends a ledger to learner $L$

1. Partition the ledger in sequences $S$ of $n$ transactions.
2. Replica $r_i \in C$ encodes $S$ into the $i$-th IDA piece $S_i$. 
The Delayed-Replication Algorithm

Idea: \( C \) sends a ledger to learner \( L \)

1. Partition the ledger in sequences \( S \) of \( n \) transactions.
2. Replica \( r_i \in C \) encodes \( S \) into the \( i \)-th IDA piece \( S_i \).
3. Replica \( r_i \in C \) sends \( S_i \) with a checksum \( C_i(S) \) of \( S \) to \( L \).
The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner $\mathcal{L}$

1. Partition the ledger in sequences $S$ of $n$ transactions.
2. Replica $r_i \in \mathcal{C}$ encodes $S$ into the $i$-th IDA piece $S_i$.
3. Replica $r_i \in \mathcal{C}$ sends $S_i$ with a checksum $C_i(S)$ of $S$ to $\mathcal{L}$.
4. $\mathcal{L}$ receives at least $n - f$ distinct pieces and decodes $S$. 

Observation ($n > 2f$)

$B = \mathcal{L} \parallel S \parallel n - f m + c \leq 2 \parallel S \parallel n + 1 + c = O(\parallel S \parallel n + c)$ bytes.

$\mathcal{L}$ receives at most $n \cdot B = O(\parallel S \parallel n + c n)$ bytes.
The Delayed-Replication Algorithm

Idea: \( C \) sends a ledger to learner \( L \)

1. Partition the ledger in sequences \( S \) of \( n \) transactions.
2. Replica \( r_i \in C \) encodes \( S \) into the \( i \)-th IDA piece \( S_i \).
3. Replica \( r_i \in C \) sends \( S_i \) with a checksum \( C_i(S) \) of \( S \) to \( L \).
4. \( L \) receives at least \( n - f \) distinct pieces and decodes \( S \).

Observation \((n > 2f)\)

- Replica \( r_i \) sends at most \( B = \left\lceil \frac{\|S\|}{n-f} \right\rceil + c \leq \frac{2\|S\|}{n} + 1 + c = O\left(\frac{\|S\|}{n} + c\right) \) bytes.
- Learner \( L \) receives at most \( n \cdot B = O(\|S\| + cn) \) bytes.
Decoding $S$ Using Simple Checksums ($n > 2f$)

- Use checksums $\text{hash}(S)$.
- The $n-f$ non-faulty replicas will provide correct pieces.
- At least $n-f > f$ messages with correct checksums.

First $x$ hashes received by $l$

- Wait until $f+1 \leq nf$ identical hashes: $\text{hash}(S)$.

- If $G$ then at least $x-f$ good hashes at most $f$ faulty hashes.

- Intensive for learners: one can choose $n-f$ out of $n$ messages in $n \cdot n - f$ ways, only one such choice is guaranteed to be correct!
Decoding $S$ Using Simple Checksums ($n > 2f$)

- Use checksums $\text{hash}(S)$.
Decoding $S$ Using Simple Checksums ($n > 2f$)

- Use checksums $\text{hash}(S)$.
- The $n - f$ non-faulty replicas will provide correct *pieces*. 
Decoding $S$ Using Simple Checksums ($n > 2f$)

- Use checksums $\text{hash}(S)$.
- The $n - f$ non-faulty replicas will provide correct pieces.
- At least $n - f > f$ messages with correct checksums.

First $x$ hashes received by $L$

- at least $x - f$ good hashes
- at most $f$ faulty hashes

Wait until $f + 1 \leq nf$ identical hashes: $\text{hash}(S)$. 

Intensive for learners: one can choose $n - f$ out of $n$ messages in $n - f$ ways only one such choice is guaranteed to be correct!
Decoding $S$ Using Simple Checksums ($n > 2f$)

- Use checksums $\text{hash}(S)$.
- The $n - f$ non-faulty replicas will provide correct pieces.
- At least $n - f > f$ messages with correct checksums.

First $x$ hashes received by $L$

- at least $x - f$ good hashes
- at most $f$ faulty hashes

Wait until $f + 1 \leq nf$ identical hashes: $\text{hash}(S)$.

- Intensive for learners: one can choose $n - f$ out of $n$ messages in $\binom{n}{n-f}$ ways only one such choice is guaranteed to be correct!
Decoding $S$ Using Tree Checksums

Use Merkle-trees to construct checksums
Consider 8 replicas and a sequence $S$.
We construct the checksum $C_5(S)$ of $S$ (used by $R_5$).

Construct a Merkle tree for pieces $S_0, \ldots, S_7$. 
Use Merkle-trees to construct checksums

Consider 8 replicas and a sequence $S$.
We construct the checksum $C_5(S)$ of $S$ (used by $R_5$).

Determine the path from root to $S_5$. 
Decoding $S$ Using Tree Checksums

**Use Merkle-trees to construct checksums**
Consider 8 replicas and a sequence $S$.
We construct the checksum $C_5(S)$ of $S$ (used by $R_5$).

Select *root* and *neighbors*: $C_5(S) = [h_4, h_{67}, h_{0123}, h_{01234567}]$. 
Theorem

Consider the learner $\mathcal{L}$, replica $\mathcal{R}$, and decided transactions $\mathcal{T}$. The delayed-replication algorithm with tree checksums guarantees

1. $\mathcal{L}$ will learn $\mathcal{T}$;
2. $\mathcal{L}$ will receive at most $|\mathcal{T}|$ messages with a total size of $O(\|\mathcal{T}\| + |\mathcal{T}| \log n)$;
3. $\mathcal{L}$ will only need at most $\frac{|\mathcal{T}|}{n}$ decode steps;
4. $\mathcal{R}$ will send at most $\frac{|\mathcal{T}|}{n}$ messages to $\mathcal{L}$ of size $O(\frac{\|\mathcal{T}\| + |\mathcal{T}| \log n}{n})$. 
Replicas typically only need the current data $V$ to decide on future updates.

- Replicas only need the full ledger $\mathcal{L}$ for recovery.
- We can use delayed-replication to reduce the data each replica has to store.

**Theorem**

The storage cost per replica can be reduced from

$$O(\|\mathcal{L}\| + \|V\|) \text{ to } O\left(\frac{\|\mathcal{L}\|}{n - f} + \frac{|\mathcal{L}|}{n} \log(n) + \|V\|\right).$$
Definition
Let \( C_1, C_2 \) be two clusters, both having non-faulty replicas.

The *cluster-sending problem* is the problem of sending a value \( v \) from \( C_1 \) to \( C_2 \) such that:

1. non-faulty replicas in \( C_2 \) receive \( v \);
2. non-faulty replicas in \( C_1 \) confirm that \( v \) was received by the non-faulty replicas in \( C_2 \);
3. replicas in \( C_2 \) only receive \( v \) if all non-faulty replicas in \( C_1 \) agree upon sending \( v \).
Definition
Let \( C_1, C_2 \) be two clusters, both having non-faulty replicas.

The \textit{cluster-sending problem} is the problem of sending a value \( v \) from \( C_1 \) to \( C_2 \) such that:

1. non-faulty replicas in \( C_2 \) \textit{receive} \( v \);
2. non-faulty replicas in \( C_1 \) \textit{confirm} that \( v \) was received by the non-faulty replicas in \( C_2 \);
3. replicas in \( C_2 \) only receive \( v \) if all non-faulty replicas in \( C_1 \) \textit{agree} upon sending \( v \).

Informal Definition
Successfully sending a value \( v \) from a cluster \( C_1 \) to a \( C_2 \) without any faulty replicas being able to \textit{disrupt sending} or send \textit{alternative forged values}. 
Basic Cluster-Sending via Broadcasting

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica in $C_1$ has a *certificate* $\text{cert}(v, C_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

$$C_1: \quad R_{1,1}, R_{1,2}, R_{1,3}, R_{1,4}, R_{1,5}$$

$$C_2: \quad R_{2,1}, R_{2,2}, R_{2,3}, R_{2,4}, R_{2,5}$$
Basic Cluster-Sending via Broadcasting

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica in $C_1$ has a *certificate* $\text{cert}(v, C_1)$ that proves agreement.
- Communication is *reliable*.
- At-most two replicas faulty in each cluster.

**Broadcast**: every replica in $C_1$ sends pairs $(v, \text{cert}(v, C_1))$ to every replica in $C_2$. 
Basic Cluster-Sending via Broadcasting

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica in $C_1$ has a *certificate* $\text{cert}(v, C_1)$ that proves agreement.
- Communication is *reliable*.
- At-most **two** replicas faulty in each cluster.

Faulty replicas can *fail* to send (in $C_1$) or to receive (in $C_2$).
Basic Cluster-Sending via Broadcasting

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica in $C_1$ has a *certificate* $\text{cert}(v, C_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

Non-faulty replicas in $C_2$ only need at-least one message $(v, \text{cert}(v, C_1))$. 
Basic Cluster-Sending via Broadcasting

Goal: send a value $v$ from cluster $C_1$ to cluster $C_2$.

Assumptions

- Every replica in $C_1$ has a *certificate* $\text{cert}(v, C_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

Replicas in $C_2$ can redistribute $(v, \text{cert}(v, C_1))$. 
Basic Cluster-Sending via Broadcasting

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica in $C_1$ has a **certificate** $\text{cert}(v, C_1)$ that proves agreement.
- Communication is **reliable**.
- At-most **two** replicas faulty in each cluster.

With certificates: a *single* message between non-faulty sender and receiver is sufficient!
Basic Cluster-Sending via Broadcasting (Without Certificates)

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica $r \in C_1$ can only *claim* agreement via a digital signature $\text{cert}(v, r)$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.
Basic Cluster-Sending via Broadcasting (Without Certificates)

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica $r \in C_1$ can only *claim* agreement via a digital signature $\text{cert}(v, r)$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

Faulty replicas can *lie* and send $\text{cert}(w, R)$ without agreement on $w$. 
Basic Cluster-Sending via Broadcasting (Without Certificates)

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica $r \in C_1$ can only *claim* agreement via a digital signature $\text{cert}(v, r)$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

Claims from *three* distinct replicas in $C_1$: at-least one from a non-faulty replica.
Basic Cluster-Sending via Broadcasting (Without Certificates)

**Goal**: send a value $v$ from cluster $C_1$ to cluster $C_2$.

**Assumptions**

- Every replica $r \in C_1$ can only *claim* agreement via a digital signature $\text{cert}(v, r)$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

Replicas in $C_2$ can redistribute $(v, \text{cert}(v, r))$. 
Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value \( v \) from cluster \( C_1 \) to cluster \( C_2 \).

Assumptions

- Every replica \( r \in C_1 \) can only claim agreement via a digital signature \( \text{cert}(v, r) \).
- Communication is reliable.
- At-most two replicas faulty in each cluster.

Without certificates: \textit{at least} \( f_{C_1} + 1 \) distinct received messages by non-faulty senders!
Efficient Cluster-Sending

Cluster-Sending via broadcasting: straightforward, *not efficient*:

- With certificates: \((f_{C_1} + 1)(f_{C_2} + 1) \approx f_{C_1} \times f_{C_2}\) messages.
- With claims: \((2f_{C_1} + 1)(f_{C_2} + 1) \approx 2f_{C_1} \times f_{C_2}\) messages.
Efficient Cluster-Sending

Cluster-Sending via broadcasting: straightforward, *not efficient*:
- With certificates: \((f_{C_1} + 1)(f_{C_2} + 1) \approx f_{C_1} \times f_{C_2}\) messages.
- With claims: \((2f_{C_1} + 1)(f_{C_2} + 1) \approx 2f_{C_1} \times f_{C_2}\) messages.

Local communication versus global communication

<table>
<thead>
<tr>
<th></th>
<th>Ping round-trip times (ms)</th>
<th>Bandwidth (Mbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OR</td>
<td>IA</td>
</tr>
<tr>
<td>Oregon</td>
<td>≤ 1</td>
<td>38</td>
</tr>
<tr>
<td>Iowa</td>
<td>≤ 1</td>
<td>33</td>
</tr>
<tr>
<td>Montreal</td>
<td>≤ 1</td>
<td>82</td>
</tr>
<tr>
<td>Belgium</td>
<td>≤ 1</td>
<td>252</td>
</tr>
<tr>
<td>Taiwan</td>
<td>≤ 1</td>
<td>137</td>
</tr>
<tr>
<td>Sydney</td>
<td>≤ 1</td>
<td></td>
</tr>
</tbody>
</table>

*Goal*: Minimize communication *between* clusters.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad f_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad f_{C_2} = 2 \]

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 messages.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad \text{f}_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad \text{f}_{C_2} = 2 \]

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 messages.

Minimize impact of faulty replicas: minimum number of messages per participant.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad f_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad f_{C_2} = 2 \]

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 messages.

Minimize impact of faulty replicas: minimum number of messages per participant.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad f_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad f_{C_2} = 2 \]

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 messages.

Minimize impact of faulty replicas: minimum number of messages per participant.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad \quad \quad f_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad \quad \quad f_{C_2} = 2 \]

Proposition (assuming certificates)
Any correct algorithm needs to send at least 14 messages.

Any \( f_{C_2} \) replicas in \( C_2 \) can be faulty: top \( f_{C_2} \) receivers receive at-least 6 messages.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad f_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad f_{C_2} = 2 \]

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 messages.
Towards a Lower-Bound for Cluster-Sending (Example)

\[ n_{C_1} = 15 \quad \text{f}_{C_1} = 7 \]
\[ n_{C_2} = 5 \quad \text{f}_{C_2} = 2 \]

Proposition (assuming certificates)
Any correct algorithm needs to send at least 14 messages.

Only \( f_{C_1} \) messages remaining, can all be sent by faulty replicas in \( C_1 \).
Towards a Lower-Bound for Cluster-Sending (Example)

\[ \begin{align*}
    n_{C_1} &= 15 \\
    n_{C_2} &= 5 \\
    f_{C_1} &= 7 \\
    f_{C_2} &= 2
\end{align*} \]

**Proposition (assuming certificates)**

Any correct algorithm needs to send at least 14 messages.
Lower-Bound for Cluster-Sending with Certificates

Basic Idea

- One message needs to be exchanged between a non-faulty sender and receiver.
- Have to deal with size imbalances between clusters.
Lower-Bound for Cluster-Sending with Certificates

Basic Idea

- One message needs to be exchanged between a non-faulty sender and receiver.
- Have to deal with size imbalances between clusters.

Theorem

Let $C_1, C_2$ be two clusters and let $\{i, j\} = \{1, 2\}$ such that $n_{C_i} \geq n_{C_j}$. Let

$$q_i = (f_{C_i} + 1) \div n_{C_j},$$

$$r_i = (f_{C_i} + 1) \mod n_{C_j},$$

$$\sigma_i = q_i n_{C_j} + r_i + f_{C_j} \text{sgn} \ r_i.$$  

Any protocol that solves the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$ needs to exchange at least $\sigma_i$ messages.
Lower-Bound for Cluster-Sending with Certificates (Example)

**Theorem**

Let $C_1, C_2$ be two clusters and let

$$q_1 = (f_{C_1} + 1) \text{ div } n_{f_{C_2}} = 7 \text{ div } 3 = 2,$$

$$r_1 = (f_{C_1} + 1) \text{ mod } n_{f_{C_2}} = 7 \text{ mod } 3 = 1,$$

$$\sigma_1 = q_1 n_{C_2} + r_1 + f_{C_2} \text{ sgn } r_1 = 2 \cdot 5 + 1 + 3 = 14.$$ 

Any protocol that solves the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$ needs to exchange at least $\sigma_1 = 14$ messages.
Lower-Bound for Cluster-Sending with Certificates (Example)

**Theorem**

Let $C_1, C_2$ be two clusters and let

\[ q_1 = (f_{C_1} + 1) \div nf_{C_2} = 7 \div 3 = 2, \]

\[ r_1 = (f_{C_1} + 1) \mod nf_{C_2} = 7 \mod 3 = 1, \]

\[ \sigma_1 = q_1 n_{C_2} + r_1 + f_{C_2} \sgn r_1 = 2 \cdot 5 + 1 + 3 = 14. \]

Any protocol that solves the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$ needs to exchange at least $\sigma_1 = 14$ messages.
Lower-Bound for Cluster-Sending with Certificates (Example)

**Theorem**

Let $C_1, C_2$ be two clusters and let

\[
q_1 = (f_{C_1} + 1) \div n_{C_2} = 7 \div 3 = 2,
\]

\[
r_1 = (f_{C_1} + 1) \mod n_{C_2} = 7 \mod 3 = 1,
\]

\[
\sigma_1 = q_1 n_{C_2} + r_1 + f_{C_2} \text{ sgn } r_1 = 2 \cdot 5 + 1 + 3 = 14.
\]

Any protocol that solves the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$ needs to exchange at least $\sigma_1 = 14$ messages.
Lower-Bound for Cluster-Sending with Certificates (Example)

**Theorem**

Let $C_1, C_2$ be two clusters and let

$$q_1 = (f_{C_1} + 1) \text{ div } nf_{C_2} = 7 \text{ div } 3 = 2,$$
$$r_1 = (f_{C_1} + 1) \text{ mod } nf_{C_2} = 7 \text{ mod } 3 = 1,$$
$$\sigma_1 = q_1 nc_2 + r_1 + f_{C_2} \text{ sgn } r_1 = 2 \cdot 5 + 1 + 3 = 14.$$

Any protocol that solves the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$ needs to exchange at least $\sigma_1 = 14$ messages.
Lower-Bound for Cluster-Sending with Claims

Basic Idea

- \( f_{C_1} + 1 \) message needs to be sent by distinct non-faulty senders to non-faulty receiver.
- Have to deal with size imbalances between clusters.

Theorem

Let \( C_1, C_2 \) be two clusters and let \( \{i, j\} = \{1, 2\} \) such that \( n_{C_i} \geq n_{C_j} \). Let

\[
\begin{align*}
q_1 &= (2f_{C_1} + 1) \div n_{f_{C_2}}, & q_2 &= (f_{C_2} + 1) \div (n_{f_{C_1}} - f_{C_1}) \\
r_1 &= (2f_{C_1} + 1) \mod n_{f_{C_2}}, & r_2 &= (f_{C_2} + 1) \mod (n_{f_{C_1}} - f_{C_1}) \\
\tau_1 &= q_1 n_{C_2} + r_1 + f_{C_2} \text{ sgn } r_1 & \tau_2 &= q_2 n_{C_1} + r_2 + 2f_{C_1} \text{ sgn } r_2.
\end{align*}
\]

Any protocol that solves the cluster-sending problem in which \( C_1 \) sends a value \( v \) to \( C_2 \) needs to exchange at least \( \tau_i \) messages.
Bijective Sending with Certificates

Assume $f_{C_1} + f_{C_2} + 1 \leq \min(n_{C_1}, n_{C_2})$.

We have $\sigma_1 = \sigma_2 = f_{C_1} + f_{C_2} + 1$.

**Protocol for the sending cluster $C_1$:**

1. All replicas in $G_{C_1}$ agree on $v$ and construct $\text{cert}(v, C_1)$.
2. Choose replicas $S_1 \subseteq C_1$ and $S_2 \subseteq C_2$ with $n_{S_2} = n_{S_1} = f_{C_1} + f_{C_2} + 1$.
3. Choose a bijection $b : S_1 \rightarrow S_2$.
4. for $R_1 \in S_1$ do
5. \hspace{1em} $R_1$ sends $(v, \text{cert}(v, C_1))$ to $b(R_1)$.

**Protocol for the receiving cluster $C_2$:**

6. event $R_2 \in G_{C_2}$ receives $(w, \text{cert}(w, C_1))$ from $R_1 \in C_1$ do
7. \hspace{1em} Broadcast $(w, \text{cert}(w, C_1))$ to all replicas in $C_2$.
8. event $R'_2 \in G_{C_2}$ receives $(w, \text{cert}(w, C_1))$ from $R_2 \in C_2$ do
9. \hspace{1em} $R'_2$ considers $w$ received.
Bijective Sending with Certificates: Example

\[ n_{C_1} = 8 \]
\[ n_{C_2} = 7 \]
\[ f_{C_1} = 3 \]
\[ f_{C_2} = 2 \]
\[ \sigma_1 = 6. \]
Bijective Sending with Certificates: Example

\[ n_{C_1} = 8 \quad \text{f}_{C_1} = 3 \]
\[ n_{C_2} = 7 \quad \text{f}_{C_2} = 2 \]
\[ \sigma_1 = 6. \]
Bijective Sending with Certificates: Example

\[ n_{C_1} = 8 \quad f_{C_1} = 3 \]
\[ n_{C_2} = 7 \quad f_{C_2} = 2 \]
\[ \sigma_1 = 6. \]
Bijective Sending with Claims

Assume $2f_{C_1} + f_{C_2} + 1 \leq \min(n_{C_1}, n_{C_2})$.

We have $\tau_1 = \tau_2 = 2f_{C_1} + f_{C_2} + 1$.

**Protocol for the sending cluster $C_1$:**
1: All replicas in $G_{C_1}$ agree on $v$.
2: Choose replicas $S_1 \subseteq C_1$ and $S_2 \subseteq C_2$ with $n_{S_2} = n_{S_1} = 2f_{C_1} + f_{C_2} + 1$.
3: Choose bijection $b : S_1 \rightarrow S_2$.
4: **for** $r_1 \in S_1$ **do**
5: $r_1$ sends $(v, \text{cert}(v, r_1))$ to $b(r_1)$.

**Protocol for the receiving cluster $C_2$:**
6: . . .
Bijective Sending with Claims

Assume $2f_{C_1} + f_{C_2} + 1 \leq \min(n_{C_1}, n_{C_2})$.

We have $\tau_1 = \tau_2 = 2f_{C_1} + f_{C_2} + 1$.

**Protocol for the sending cluster $C_1$:**

1: ....

**Protocol for the receiving cluster $C_2$:**

6: **event** $r_2 \in G_{C_2}$ receives $(w, \text{cert}(w, r'_1))$ from $r'_1 \in C_1$ do

7: Broadcast $(w, \text{cert}(w, r'_1))$ to all replicas in $C_2$.

8: **event** $r'_2 \in G_{C_2}$ receives $f_{C_1} + 1$ messages $(w, \text{cert}(w, r'_1))$:  

   (i) each message is sent by a replica in $C_2$;  
   (ii) each message carries the same value $w$; and  
   (iii) each message has a distinct signature $\text{cert}(w, r'_1), r'_1 \in C_1$

   do

9: $r'_2$ considers $w$ received.
Generalizing Bijective Sending

Consider bijective sending from $C_1$ to $C_2$, $n_{C_1} \geq \sigma_1 > n_{C_2}$, with certificates.

- Bijective sending requires $f_{C_1} + f_{C_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.
Generalizing Bijective Sending

Consider bijective sending from $C_1$ to $C_2$, $n_{C_1} \geq \sigma_1 > n_{C_2}$, with certificates.

- Bijective sending requires $f_{C_1} + f_{C_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending
Generalizing Bijective Sending

Consider bijective sending from $C_1$ to $C_2$, $n_{C_1} \geq \sigma_1 > n_{C_2}$, with certificates.

- Bijective sending requires $f_{C_1} + f_{C_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition $\sigma_1$ replicas of $C_1$ into $n_{C_2}$-sized clusters.
Generalizing Bijective Sending

Consider bijective sending from $C_1$ to $C_2$, $n_{C_1} \geq \sigma_1 > n_{C_2}$, with certificates.

- Bijective sending requires $f_{C_1} + f_{C_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition $\sigma_1$ replicas of $C_1$ into $n_{C_2}$-sized clusters.
- Bijective send from each cluster in the partition to $C_2$. 

![Diagram of bijective sending](image)

- $C_1$: $R_{1,1}, R_{1,2}, R_{1,3}, R_{1,4}, R_{1,5}, R_{1,6}, R_{1,7}, R_{1,8}, R_{1,9}, R_{1,10}, R_{1,11}, R_{1,12}, R_{1,13}, R_{1,14}, R_{1,15}$
- $C_2$: $R_{2,1}, R_{2,2}, R_{2,3}, R_{2,4}, R_{2,5}$
- $P_1$, $P_2$, and $P'$ represent partitions of $C_1$ into clusters of size $n_{C_2}$.
Consider bijective sending from $C_1$ to $C_2$, $n_{C_1} \geq \sigma_1 > n_{C_2}$, with certificates.  
- Bijective sending requires $f_{C_1} + f_{C_2} + 1$ distinct replicas in both clusters.  
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition $\sigma_1$ replicas of $C_1$ into $n_{C_2}$-sized clusters.  
- Bijective send from each cluster in the partition to $C_2$.  
- $n_{C_1} \geq \sigma_1$ holds always if $n_{C_1} > 3f_{C_1}$ and $n_{C_2} > 3f_{C_2}$.
Partitioned Bijective Sending

Corollary

Consider the cluster-sending problem in which $C_1$ sends a value $v$ to $C_2$.

1. If $n_C > 3f_C$ for all clusters $C$ and replicas only have crash failures or omit failures, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.

2. If $n_C > 3f_C$ for all clusters $C$ and clusters can produce certificates, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.

3. If $n_C > 4f_C$ for all clusters $C$ and replicas can digitally sign claims, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.

These protocols solve cluster-sending using $O(\max(n_{C_1}, n_{C_2}))$ messages of size $O(\|v\|)$ each.
Cluster-sending: Can we do Better?

**Pessimistic**

*No*: these algorithms are worst-case optimal. Cannot do better than *linear communication* in the size of the clusters.
Cluster-sending: Can we do Better?

Pessimistic

No: these algorithms are worst-case optimal. Cannot do better than *linear communication* in the size of the clusters.

Probabilistic

Yes: if we randomly choose sender and receiver, then we often do much better! Probabilistic approach: expected-case only *constant communication* (four steps).
Motivation: High-Performance Resilient Systems

Partition the system: More storage and potentially more performance.
Potentially lower latencies if data ends up closer to users.
Adding shards $\implies$ adding throughput (parallel processing), adding storage.
Motivation: High-Performance Resilient Systems

Single System
(All Data)

European Node
(European Data)

American Node
(American Data)

 Requests
(All Data)

Requests
(Mixed Data)

(coordination)

Requests
(European Data)

Requests
(American Data)

Resilient system
Motivation: High-Performance Resilient Systems

Resilient system

> Individual shards are consensus-operated *blockchains*. 
Motivation: High-Performance Resilient Systems

Resilient system

- Individual shards are consensus-operated blockchains.
- Communication between shards via cluster-sending.
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;
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.

Given a transaction $\tau$, a DBMS maintains

**Atomicity.** Either all or none of the operations of $\tau$ are reflected in the database.

**Consistency** Execution of $\tau$ in isolation preserves data consistency.
E.g., integrity constraints—this is stronger than CAP-Consistency.

**Isolation** $\tau$ is “unaware” of other transactions executing concurrently
“As-if” all transactions are executed in a sequential order.

**Durability** After $\tau$ completes successfully, the changes $\tau$ made persist.
If $\tau$ fails, then no changes persist due to atomicity.
The ACID Properties

Contract between a DBMS and its users.

Given a transaction $\tau$, a DBMS maintains

**Atomicity.** Either all or none of the operations of $\tau$ are reflected in the database.

**Consistency** Execution of $\tau$ in *isolation* preserves data consistency.
   E.g., integrity constraints—this is stronger than CAP-Consistency.

**Isolation** $\tau$ is “unaware” of other transactions executing concurrently
   “As-if” all transactions are executed in a sequential order.

**Durability** After $\tau$ completes successfully, the changes $\tau$ made persist.
   If $\tau$ fails, then *no* changes persist due to atomicity.

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

Contract between a DBMS and its users.

Given a transaction \( \tau \), a DBMS maintains

**Atomicity.** Either all or none of the operations of \( \tau \) are reflected in the database.

**Consistency** Execution of \( \tau \) in isolation preserves data consistency.

E.g., integrity constraints—this is stronger than CAP-Consistency.

**Isolation** \( \tau \) is “unaware” of other transactions executing concurrently

“As-if” all transactions are executed in a sequential order.

**Durability** After \( \tau \) completes successfully, the changes \( \tau \) made persist.

If \( \tau \) fails, then no changes persist due to atomicity.

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.

Given a transaction $\tau$, a DBMS maintains

**Atomicity.** Either all or none of the operations of $\tau$ are reflected in the database.

**Consistency** Execution of $\tau$ in isolation preserves data consistency.
   E.g., integrity constraints—this is stronger than CAP-Consistency.

**Isolation** $\tau$ is “unaware” of other transactions executing concurrently
   “As-if” all transactions are executed in a sequential order.

**Durability** After $\tau$ completes successfully, the changes $\tau$ made persist.
   If $\tau$ fails, then no changes persist due to atomicity.

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.
Background on Resilience
Consider a transaction $\tau$ requested by client $c$ in a resilient system.
Background on Resilience

Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.
Background on Resilience

Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.

Non-sharded resilient systems

- Consensus solves all of the above.
- In particular replication order is execution order.
- Consecutive execution guarantees ACID.
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The current state is the balance of each account obtained by executing transactions.
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts
The *current state* is the balance of each account obtained by executing transactions.

\[\begin{align*}
\tau_1 &= \text{“add $500 to Ana”}; \\
\tau_2 &= \text{“add $200 to Bo and $300 to Elisa”}; \\
\tau_3 &= \text{“move $30 from Ana to Elisa”}; \\
\tau_4 &= \text{“remove $70 from Elisa”};
\end{align*}\]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts
The *current state* is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add } 500 \text{ to } \textit{Ana}; \]
\[ \tau_2 = \text{“add } 200 \text{ to } \textit{Bo} \text{ and } 300 \text{ to } \textit{Elisa}; \]
\[ \tau_3 = \text{“move } 30 \text{ from } \textit{Ana} \text{ to } \textit{Elisa}; \]
\[ \tau_4 = \text{“remove } 70 \text{ from } \textit{Elisa}; \]

<table>
<thead>
<tr>
<th></th>
<th>Ana</th>
<th>Bo</th>
<th>Elisa</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>$500$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>$0$</td>
<td>$200$</td>
<td>$300$</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>$470$</td>
<td>$200$</td>
<td>$330$</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>$470$</td>
<td>$200$</td>
<td>$260$</td>
</tr>
</tbody>
</table>
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts
The *current state* is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add $500 to Ana”}; \]
\[ \tau_2 = \text{“add $200 to Bo and $300 to Elisa”}; \]
\[ \tau_3 = \text{“move $30 from Ana to Elisa”}; \]
\[ \tau_4 = \text{“remove $70 from Elisa”}; \]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts
The current state is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add $500 to Ana”}; \]
\[ \tau_2 = \text{“add $200 to Bo and $300 to Elisa”}; \]
\[ \tau_3 = \text{“move $30 from Ana to Elisa”}; \]
\[ \tau_4 = \text{“remove $70 from Elisa”}; \]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts
The *current state* is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add $500 to Ana”}; \]
\[ \tau_2 = \text{“add $200 to Bo and $300 to Elisa”}; \]
\[ \tau_3 = \text{“move $30 from Ana to Elisa”}; \]
\[ \tau_4 = \text{“remove $70 from Elisa”}; \]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

\[
\begin{align*}
\tau_1 &= \text{“add $500 to Ana”;} \\
\tau_2 &= \text{“add $200 to Bo and $300 to Elisa”;} \\
\tau_3 &= \text{“move $30 from Ana to Elisa”;} \\
\tau_4 &= \text{“remove $70 from Elisa”;} \\
\end{align*}
\]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add $500 to Ana”}; \]
\[ \tau_2 = \text{“add $200 to Bo and $300 to Elisa”}; \]
\[ \tau_3 = \text{“move $30 from Ana to Elisa”}; \]
\[ \tau_4 = \text{“remove $70 from Elisa”}; \]
\[ \tau_5 = \text{“move $500 from Ana to Bo”}. \]
Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

\[ \tau_1 = \text{“add $500 to Ana”;} \]
\[ \tau_2 = \text{“add $200 to Bo and $300 to Elisa”;} \]
\[ \tau_3 = \text{“move $30 from Ana to Elisa”;} \]
\[ \tau_4 = \text{“remove $70 from Elisa”;} \]
\[ \tau_5 = \text{aborted} \text{ (would invalidate balances).} \]

<table>
<thead>
<tr>
<th></th>
<th>Ana</th>
<th>Bo</th>
<th>Elisa</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$0</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>$500</td>
<td>$0</td>
<td>$0</td>
<td>$300</td>
</tr>
<tr>
<td>$500</td>
<td>$200</td>
<td>$0</td>
<td>$70</td>
</tr>
</tbody>
</table>

\[ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4 \rightarrow \tau_5 \]
Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.
Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.

$\tau$ must be replicated among all replicas of all shards affected by $\tau$!
Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in *five* steps

1. $\tau$ needs to be *received* by the system;
2. $\tau$ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order for $\tau$*;
4. the replicas each need to *execute* $\tau$ and *update* their current state accordingly;
5. the client $c$ needs to be *informed* about the result.

What is a consistent execution order *across* shards? Does it relate to the *replication order*?
Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.

Dependencies on data in other shards? Writes to data in other shards?
Consider a transaction $\tau$ requested by client $c$ in a resilient system.

$\tau$ is processed in five steps

1. $\tau$ needs to be received by the system;
2. $\tau$ must be replicated among all replicas in the system;
3. the replicas need to agree on an execution order for $\tau$;
4. the replicas each need to execute $\tau$ and update their current state accordingly;
5. the client $c$ needs to be informed about the result.

A single consensus does no longer solve all of the above!
Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system
Say we have 26 shards: $C_a, C_b, \ldots, C_z$,
such that shard $C_\xi$ holds accounts of people whose name starts with $\xi$. 
Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system
Say we have 26 shards: $C_a, C_b, \ldots, C_z,$
such that shard $C_\xi$ holds accounts of people whose name starts with $\xi$.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction $\tau$. 
Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system
Say we have 26 shards: $C_a, C_b, \ldots, C_z$,
such that shard $C_\xi$ holds accounts of people whose name starts with $\xi$.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction $\tau$.

$\tau_1 =$ “add $500 to Ana”,
$\tau_2 =$ “add $200 to Bo and $300 to Elisa”,
$\tau_3 =$ “move $30 from Ana to Elisa”; 
$\tau_4 =$ “remove $70 from Elisa”,}
Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system
Say we have 26 shards: $C_a, C_b, \ldots, C_z$, such that shard $C_\xi$ holds accounts of people whose name starts with $\xi$.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction $\tau$.

\[
\begin{align*}
\tau_1 &= \text{“add $500 to Ana”}, & \text{shards}(\tau_1) &= \{C_a\}; \\
\tau_2 &= \text{“add $200 to Bo and $300 to Elisa”}, & \text{shards}(\tau_2) &= \{C_b, C_e\}; \\
\tau_3 &= \text{“move $30 from Ana to Elisa”}, & \text{shards}(\tau_3) &= \{C_a, C_e\}; \\
\tau_4 &= \text{“remove $70 from Elisa”}, & \text{shards}(\tau_4) &= \{C_e\}.
\end{align*}
\]
Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $C_a, C_b, \ldots, C_z$, such that shard $C_\xi$ holds accounts of people whose name starts with $\xi$.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction $\tau$.

$\tau_1 =$ “add $500 to Ana”, \hspace{1cm} \text{shards}(\tau_1) = \{C_a\}; \hspace{1cm} \text{(single-shard)}$

$\tau_2 =$ “add $200 to Bo and $300 to Elisa”, \hspace{1cm} \text{shards}(\tau_2) = \{C_b, C_e\}; \hspace{1cm} \text{(multi-shard)}$

$\tau_3 =$ “move $30 from Ana to Elisa”; \hspace{1cm} \text{shards}(\tau_3) = \{C_a, C_e\}; \hspace{1cm} \text{(multi-shard)}$

$\tau_4 =$ “remove $70 from Elisa”, \hspace{1cm} \text{shards}(\tau_4) = \{C_e\}. \hspace{1cm} \text{(single-shard)}$
An Example of Concurrent Execution

Consider a banking example in which

- Bo wants to transfer $400 to Ana
  \textit{if} Ana has at-least $100 and Bo has at-least $700,
- Ana wants to transfer $300 to Elisa
  \textit{if} Ana has at-least $500,

and no account is allowed to have a negative balance.
An Example of Concurrent Execution

Consider a banking example in which

- Bo wants to transfer $400 to Ana
  - if Ana has at-least $100 and Bo has at-least $700,
- Ana wants to transfer $300 to Elisa
  - if Ana 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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$100$</td>
</tr>
<tr>
<td>$B$</td>
<td>$300$</td>
</tr>
<tr>
<td>$E$</td>
<td>$0$</td>
</tr>
</tbody>
</table>
An Example of Concurrent Execution

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

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$100</td>
<td>$300</td>
<td>$0</td>
</tr>
</tbody>
</table>

\( \tau_1 \) at \( C_a \):

\( A \geq 100? \)

\( A := A + 400 \)

\[ \begin{array}{c|c|c|c|c}
   & A & B & E \\
---&---&---&---
   A  & $500 & $300 & $0 \\
   B  & $300 & & \\
   E  & & & \end{array} \]
An Example of Concurrent Execution

\[ \tau_1 = A \geq 100?, \ A := A + 400, \ B \geq 700?, \ B := B - 400; \]
\[ \tau_2 = A \geq 500?, \ A := A - 300, \ E := E + 300. \]
An Example of Concurrent Execution

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

\[ \tau_1 = A \geq 100?, \ A := A + 400, \ B \geq 700?, \ B := B - 400; \]
\[ \tau_2 = A \geq 500?, \ A := A - 300, \ E := E + 300. \]
An Example of Concurrent Execution

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

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100$</td>
<td>$300$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

\(\tau_1\) at \(C_a\):

\[ A \geq 100? \]
\[ A := A + 400 \]

\(\tau_1\) at \(C_a\):

\[ A \geq 500? \]
\[ A := A - 300 \]

\(\tau_1\) at \(C_b\):

\[ B \geq 700? \]

\(\tau_2\) at \(C_a\):

\[ A \geq 100? \]
\[ A := A + 400 \]

\(\tau_2\) at \(C_a\):

\[ A \geq 500? \]
\[ A := A - 300 \]

\(\tau_1\) at \(C_a\):

\[ A := A - 400 \]

\(\tau_2\) at \(C_e\):

\[ E := E + 300 \]
An Example of Concurrent Execution–Revisited

Consider a banking example in which

- Bo wants to transfer $400 to Ana
  *if* Ana has at-least $100 and Bo has at-least $700,
- Ana wants to transfer $300 to Elisa
  *if* Ana 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 Ana 
  \[ \text{if } \text{Ana has at-least } 100 \text{ and Bo has at-least } 700, \]
- Ana wants to transfer $300 to Elisa
  \[ \text{if } \text{Ana has at-least } 500, \]
and no account is allowed to have a negative balance.

\[
\begin{align*}
\tau_1 &= A \geq 100?, \ A := A + 400, \ B \geq 700?, \ B := B - 400; \\
\tau_2 &= A \geq 500?, \ A := A - 300, \ E := E + 300.
\end{align*}
\]

Transactions \( \tau_1 \) and \( \tau_2 \) 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 Ana
  if Ana has at-least $100 and Bo has at-least $700,
- Ana wants to transfer $300 to Elisa
  if Ana 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 \( \tau_1 \) and \( \tau_2 \) make sense: their isolated execution will never make balances negative.

Guarantee by an ACID-compliant system

No account will ever have a negative balance.
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).
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.
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).
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.
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).
Consider the transaction $\tau$: $\tau = \text{“if Ana has$500 and Bo has$200, then move$400 from Ana to Elisa; move$100 from Bo to Elisa”.}
Consider the transaction $\tau$: 

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

What are the operations of $\tau$?

Depending on how the system executes $\tau$ and the database state:

- Might read from $Ana$’s account.
- Might read from $Bo$’s account.
- Might write to $Ana$’s account.
- Might write to $Bo$’s account.
- Might write to $Elisa$’s account.
Simplified Transaction Notation

Consider the transaction $\tau$: 

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

Simplifying assumption

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

Consider the transaction $\tau$:

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

Simplifying assumption

Each transaction is a sequence of read and write operations ending in commit or abort. $\text{Read}_\tau(Ana), \text{Read}_\tau(Bo), \text{Write}_\tau(Ana), \text{Write}_\tau(Bo), \text{Read}_\tau(Elisa), \text{Write}_\tau(Elisa), \text{Commit}_\tau$.  

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: \( \tau_1 \), then \( \tau_2 \) (insufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$100</td>
</tr>
<tr>
<td>( B )</td>
<td>$300</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>
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: \( \tau_1 \), then \( \tau_2 \) (insufficient funds)
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: \( \tau_1 \), then \( \tau_2 \) (insufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>( \text{Read}_{\tau_1}(A) )</th>
<th>( \text{Write}_{\tau_1}(A) )</th>
<th>( \text{Read}_{\tau_1}(B) )</th>
<th>( \text{Write}_{\tau_1}(A) )</th>
<th>( \text{Abort}_{\tau_1} )</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
<td>$300</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( B )</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E )</td>
</tr>
</tbody>
</table>

\( \text{Schedule} \)
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: \( \tau_1 \), then \( \tau_2 \) (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$100</td>
</tr>
<tr>
<td>( B )</td>
<td>$800</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>
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: $\tau_1$, then $\tau_2$ (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ $100$</td>
<td>Read$<em>{\tau_1}(A)$, Write$</em>{\tau_1}(A)$, Read$<em>{\tau_1}(B)$, Write$</em>{\tau_1}(B)$, Commit$_{\tau_1}$</td>
</tr>
<tr>
<td>$B$ $800$</td>
<td>Read$<em>{\tau_2}(A)$, Write$</em>{\tau_2}(A)$, Read$<em>{\tau_2}(E)$, Write$</em>{\tau_2}(E)$, Commit$_{\tau_2}$</td>
</tr>
<tr>
<td>$E$ $0$</td>
<td></td>
</tr>
</tbody>
</table>

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: \( \tau_1 \), then \( \tau_2 \) (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(\tau_1)(A) | Write(\tau_1)(A) | Read(\tau_1)(B) | Write(\tau_1)(B) | Commit(\tau_1)</td>
</tr>
<tr>
<td>( A )</td>
<td>$100</td>
</tr>
<tr>
<td>( B )</td>
<td>$800</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance (final)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$200</td>
</tr>
<tr>
<td>( B )</td>
<td>$400</td>
</tr>
<tr>
<td>( E )</td>
<td>$300</td>
</tr>
</tbody>
</table>
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: \( \tau_2 \), then \( \tau_1 \) (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$100</td>
</tr>
<tr>
<td>( B )</td>
<td>$800</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>
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: \( \tau_2 \), then \( \tau_1 \) (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(_{\tau_1}(A))</td>
</tr>
<tr>
<td>( A )</td>
<td>$100</td>
</tr>
<tr>
<td>( B )</td>
<td>$800</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Read(_{\tau_1}(B))</th>
<th>Write(_{\tau_1}(B))</th>
<th>Commit(_{\tau_1})</th>
</tr>
</thead>
</table>
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: \( \tau_2 \), then \( \tau_1 \) (Bob has sufficient funds)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Instance (initial)</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(\tau_2(A))</td>
<td>A $500</td>
</tr>
<tr>
<td></td>
<td>Abort(\tau_2)</td>
<td></td>
</tr>
<tr>
<td>Read(\tau_1(A))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(\tau_1(A))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(\tau_1(B))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(\tau_1(B))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit(\tau_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>$100</td>
<td>A $500</td>
</tr>
<tr>
<td>B</td>
<td>$800</td>
<td>B $400</td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
<td>E $0</td>
</tr>
</tbody>
</table>
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: \( \tau_2 \), then \( \tau_1 \) (Ana has sufficient funds)

Instance (initial)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$500</td>
</tr>
<tr>
<td>( B )</td>
<td>$300</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
</tr>
</tbody>
</table>
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: \( \tau_2, \) then \( \tau_1 \) (Ana has sufficient funds)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>( \tau_1 )</th>
<th>( \tau_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$500</td>
<td>( \text{Read}_{\tau_2}(A) )</td>
</tr>
<tr>
<td>( B )</td>
<td>$300</td>
<td>( \text{Write}_{\tau_2}(A) )</td>
</tr>
<tr>
<td>( E )</td>
<td>$0</td>
<td>( \text{Read}_{\tau_2}(E) )</td>
</tr>
</tbody>
</table>

Commit \( \tau_2 \)

Read \( \tau_1 \) (\( A \))

Write \( \tau_1 \) (\( A \))

Abort \( \tau_1 \)
An Example of Schedules

Consider again the transactions

\[
\begin{align*}
\tau_1 &= A \geq 100?, \ A := A + 400, \ B \geq 700?, \ B := B - 400; \\
\tau_2 &= A \geq 500?, \ A := A - 300, \ E := E + 300.
\end{align*}
\]

Serial schedule: \(\tau_2\), then \(\tau_1\) (Ana has sufficient funds)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Instance (initial)</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_2)</td>
<td>Read(<em>{\tau_2}(A)) \nWrite(</em>{\tau_2}(A)) \nRead(<em>{\tau_2}(E)) \nWrite(</em>{\tau_2}(E)) \nCommit(_{\tau_2})</td>
<td>(\tau_1)</td>
</tr>
<tr>
<td>(A)</td>
<td>$500</td>
<td>(A)</td>
</tr>
<tr>
<td>(B)</td>
<td>$300</td>
<td>(B)</td>
</tr>
<tr>
<td>(E)</td>
<td>$0</td>
<td>(E)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$100</td>
</tr>
<tr>
<td>B</td>
<td>$300</td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>$100 \</td>
</tr>
<tr>
<td>( B )</td>
<td>$300 \</td>
</tr>
<tr>
<td>( E )</td>
<td>$0 \</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \tau_1 )</th>
<th>( \tau_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Read}_{\tau_1}(A) )</td>
<td>( \text{Read}_{\tau_2}(A) )</td>
</tr>
<tr>
<td>( \text{Write}_{\tau_1}(A) )</td>
<td>( \text{Write}_{\tau_2}(A) )</td>
</tr>
<tr>
<td>( \text{Read}_{\tau_1}(B) )</td>
<td>( \text{Read}_{\tau_2}(E) )</td>
</tr>
<tr>
<td>( \text{Write}_{\tau_1}(A) )</td>
<td>( \text{Write}_{\tau_2}(E) )</td>
</tr>
</tbody>
</table>
| \( \text{Commit}_{\tau_2} \) | \n
| \( \text{Abort}_{\tau_1} \) | \n
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

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Instance (initial)</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>( \text{Read}<em>{\tau_1}(A) ) ( \text{Write}</em>{\tau_1}(A) )</td>
<td>( A ) $100</td>
</tr>
<tr>
<td></td>
<td>( \text{Read}<em>{\tau_1}(B) ) ( \text{Read}</em>{\tau_1}(A) ) ( \text{Write}<em>{\tau_1}(A) ) ( \text{Abort}</em>{\tau_1} )</td>
<td>( B ) $300</td>
</tr>
<tr>
<td></td>
<td>( \text{Read}<em>{\tau_2}(A) ) ( \text{Write}</em>{\tau_2}(A) ) ( \text{Read}<em>{\tau_2}(E) ) ( \text{Write}</em>{\tau_2}(E) ) ( \text{Commit}_{\tau_2} )</td>
<td>( E ) $0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Instance (initial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( E )</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_{\tau_1}(A)</td>
</tr>
<tr>
<td>Write_{\tau_2}(A)</td>
</tr>
<tr>
<td>Write_{\tau_2}(E)</td>
</tr>
<tr>
<td>Write_{\tau_1}(A)</td>
</tr>
<tr>
<td>Write_{\tau_1}(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance (initial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>$500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_{\tau_1}(A)</td>
</tr>
<tr>
<td>Write_{\tau_2}(A)</td>
</tr>
<tr>
<td>Write_{\tau_2}(E)</td>
</tr>
<tr>
<td>Write_{\tau_1}(A)</td>
</tr>
<tr>
<td>Write_{\tau_1}(B)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_{\tau_1}(A)</td>
<td>A $900</td>
</tr>
<tr>
<td>Write_{\tau_2}(A)</td>
<td>B $400</td>
</tr>
<tr>
<td>Read_{\tau_2}(A)</td>
<td>E $300</td>
</tr>
<tr>
<td>Write_{\tau_2}(E)</td>
<td></td>
</tr>
<tr>
<td>Commit_{\tau_2}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>A $500</th>
<th>B $800</th>
<th>E $0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>$</th>
<th>$500</th>
<th>$800</th>
<th>$0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td></td>
<td>$500$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>$800$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Example of Schedules

Consider again the transactions

\[ \tau_1 = A \geq 100, \quad A := A + 400, \quad B \geq 700, \quad B := B - 400; \]
\[ \tau_2 = A \geq 500, \quad A := A - 300, \quad E := E + 300. \]

Non-serial schedule—A third example

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_{\tau_1}(A)</td>
</tr>
<tr>
<td>Write_{\tau_1}(A)</td>
</tr>
<tr>
<td>Read_{\tau_1}(B)</td>
</tr>
<tr>
<td>Write_{\tau_1}(B)</td>
</tr>
<tr>
<td>Commit_{\tau_1}</td>
</tr>
<tr>
<td>Read_{\tau_2}(A)</td>
</tr>
<tr>
<td>Write_{\tau_2}(A)</td>
</tr>
<tr>
<td>Read_{\tau_2}(E)</td>
</tr>
<tr>
<td>Write_{\tau_2}(E)</td>
</tr>
<tr>
<td>Commit_{\tau_2}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance (initial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>
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

\[
\begin{array}{c|c|c|c|c}
\text{Instance} & \text{Read}_{\tau_1}(A) & \text{Write}_{\tau_1}(A) & \text{Read}_{\tau_2}(A) & \text{Write}_{\tau_2}(A) \\
(\text{initial}) & \text{Write}_{\tau_1}(B) & \text{Read}_{\tau_1}(B) & \text{Commit}_{\tau_1} & \text{Commit}_{\tau_2} \\
A & \$500 & & & \\
B & \$800 & & & \\
E & \$0 & & & \\
\hline
\text{Instance} & \text{A} & \text{B} & \text{E} \\
(\text{final}) & \$200 & \$400 & \$300
\end{array}
\]
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)

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$500$</td>
</tr>
<tr>
<td>$B$</td>
<td>$800$</td>
</tr>
<tr>
<td>$E$</td>
<td>$0$</td>
</tr>
</tbody>
</table>
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)

\[
\begin{array}{c|c|c}
\text{Instance} & \text{Read} & \text{Write} \\
\text{(initial)} & \tau_1(A) & \tau_1(A) \\
A & 500 & \\
B & 800 & \\
E & 0 & \\
\text{Schedule} & \tau_2(A) & \tau_2(A) \\
\text{Read} & \tau_2(E) & \tau_2(E) \\
\text{Write} & \tau_1(B) & \tau_1(B) \\
\text{Commit} & \tau_1 & \tau_2 \\
\text{(final)} & 600 & 400 \\
A & 800 & 0 \\
B & 400 & \\
E & 300 & \\
\end{array}
\]
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)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Instance (initial)</th>
<th>Instance (final)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( A )</td>
<td>$500</td>
</tr>
<tr>
<td></td>
<td>( B )</td>
<td>$800</td>
</tr>
<tr>
<td></td>
<td>( E )</td>
<td>$0</td>
</tr>
<tr>
<td>Read( \tau_1 )(A)</td>
<td>Write( \tau_1 )(A)</td>
<td>Read( \tau_2 )(A)</td>
</tr>
<tr>
<td>Read( \tau_1 )(B)</td>
<td>Write( \tau_1 )(B)</td>
<td>Read( \tau_2 )(E)</td>
</tr>
<tr>
<td>Commit( \tau_1 )</td>
<td></td>
<td>Commit( \tau_2 )</td>
</tr>
</tbody>
</table>
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.
Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- 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.
- 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.
- 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.
- 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.
▶ 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 system*, executes, *releases the system*?
Guaranteeing Isolation

Simplified point-of-view

▶ A transaction is a *thread* in a multi-threaded program.
▶ 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 system*, executes, *releases the system*?

This will enforce a *serial schedule*. 
Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- 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 system*, executes, *releases the system*?

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., accounts, tables, rows, ....
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects.
E.g., accounts, tables, rows, ....

In our examples we abstract from details: accounts are database objects.
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ...

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

Using fine-grained locks

A transaction $\tau$ 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., accounts, tables, rows, ....

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

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

![Schedule Diagram]

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Read_{\tau_1}(A)</th>
<th>Write_{\tau_1}(A)</th>
<th>Commit_{\tau_1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance (final)</th>
<th>Read_{\tau_2}(A)</th>
<th>Write_{\tau_2}(A)</th>
<th>Read_{\tau_2}(E)</th>
<th>Write_{\tau_2}(E)</th>
<th>Commit_{\tau_2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$900</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$400</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$300</td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, …

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

Lock-based access solves some issues …

Schedule

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Lock$_{\tau_1}(A)$</th>
<th>Read$_{\tau_1}(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$500</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>$800</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
<td></td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*. E.g., accounts, tables, rows, …. In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues …

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$500$</td>
</tr>
<tr>
<td>$B$</td>
<td>$800$</td>
</tr>
<tr>
<td>$E$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

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

$\text{Lock}_{\tau_2}(A)$
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ….

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

Lock-based access solves some issues …

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$500</td>
</tr>
<tr>
<td>B</td>
<td>$800</td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*. E.g., accounts, tables, rows, ...

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

Lock-based access solves *some* issues …

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock$_{\tau_1}(A)$</td>
</tr>
<tr>
<td>Read$_{\tau_1}(A)$</td>
</tr>
<tr>
<td>Lock$_{\tau_2}(A)$</td>
</tr>
<tr>
<td>Write$_{\tau_1}(A)$</td>
</tr>
<tr>
<td>Release$_{\tau_1}(A)$</td>
</tr>
<tr>
<td>Read$_{\tau_2}(A)$</td>
</tr>
</tbody>
</table>

**Instance (initial)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$500</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>$800</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
<td></td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ....

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

Lock-based access solves some issues ...

Instance (initial)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$500</td>
</tr>
<tr>
<td>B</td>
<td>$800</td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
</tr>
</tbody>
</table>

Schedule

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Lock}_{\tau_1}(A)$</td>
<td>$\text{Lock}_{\tau_2}(A)$</td>
</tr>
<tr>
<td>$\text{Read}_{\tau_1}(A)$</td>
<td>$\text{Read}_{\tau_2}(A)$</td>
</tr>
<tr>
<td>$\text{Write}_{\tau_1}(A)$</td>
<td>$\text{...}$</td>
</tr>
<tr>
<td>$\text{Release}_{\tau_1}(A)$</td>
<td>$\text{Commit}_{\tau_2}$</td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on \textit{database objects}.
E.g., accounts, tables, rows, ….

In our examples we abstract from details: \textit{accounts} are database objects.

Lock-based access solves \textit{some} issues …

\textbf{Schedule}

\begin{itemize}
  \item Lock$_{\tau_1}(A)$
  \item Read$_{\tau_1}(A)$
  \item Write$_{\tau_1}(A)$
  \item Release$_{\tau_1}(A)$
  \item Commit$_{\tau_1}$
  \item Lock$_{\tau_2}(A)$
  \item Read$_{\tau_2}(A)$
  \item \ldots
  \item Commit$_{\tau_2}$
\end{itemize}
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, …

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

Lock-based access solves some issues …

\[
\begin{array}{c|c}
\text{Instance (initial)} & A & \text{initial} \\
\hline
A & $500 \\
B & $800 \\
E & $0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{Schedule} & \text{Instance (final)} \\
\hline
\text{Lock}_{\tau_1}(A) & A & $600 \\
\text{Read}_{\tau_1}(A) & B & $400 \\
\text{Write}_{\tau_1}(A) & E & $300 \\
\text{Release}_{\tau_1}(A) & \text{Commit}_{\tau_2} & \\
\text{Commit}_{\tau_1} & \end{array}
\]
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects.
E.g., accounts, tables, rows, …

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

…but not all issues …

<table>
<thead>
<tr>
<th>Instance (initial)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$100</td>
</tr>
<tr>
<td>B</td>
<td>$300</td>
</tr>
<tr>
<td>E</td>
<td>$0</td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*. E.g., accounts, tables, rows, . . .

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

...but not *all* issues ...
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, …. 

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

…but not all issues …
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ....

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

...and introduces new issues.

Consider two transactions that both want to access Ana and Bo:

\[ \tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \ldots; \]
\[ \tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_1}(A), \ldots \]
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ….

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

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

\[ \tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \ldots; \quad \tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_1}(A), \ldots \]

Schedule:

<table>
<thead>
<tr>
<th>Lock_{\tau_1}(A)</th>
<th>Lock_{\tau_2}(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_{\tau_1}(B)</td>
<td>Lock_{\tau_2}(A)</td>
</tr>
</tbody>
</table>
Improving Isolation using Locks

Idea: Use a fine-grained set of locks on database objects. E.g., accounts, tables, rows, ....

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

...and introduces new issues.
Consider two transactions that both want to access Ana and Bo:

\[
\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \ldots; \quad \tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_1}(A), \ldots
\]

Schedule

<table>
<thead>
<tr>
<th>\text{Lock}_{\tau_1}(A)</th>
<th>\text{Lock}_{\tau_2}(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Lock}_{\tau_1}(B)</td>
<td>\text{Lock}_{\tau_2}(A)</td>
</tr>
</tbody>
</table>

Both transactions will wait forever: a deadlock!
Locking itself does not guarantee *serializability*.

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

**Two-phase locking protocol (2PL)**

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

- *Growing phase* during which execution can obtain locks, and not release them;
- *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 ($\text{Commit}_\tau$ or $\text{Abort}_\tau$).

*Notice—Nothing to deal with deadlocks.*
Achieving Serializability with Locks

Locking itself does not guarantee *serializability*. Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

Two-phase locking protocol (2PL)
Execution of transaction $\tau$ 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 $\text{Lock}_{\tau}(O)$. Notice—Nothing to deal with deadlocks.
Achieving Serializability with Locks

Locking itself does not guarantee serializability.

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

Two-phase locking protocol (2PL)
Execution of transaction $\tau$ 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 ($\text{Commit}_{\tau}$ or $\text{Abort}_{\tau}$).
Achieving Serializability with Locks

Locking itself does not guarantee *serializability*. Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

**Two-phase locking protocol (2PL)**

Execution of transaction $\tau$ 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 ($\text{Commit}_\tau$ or $\text{Abort}_\tau$).

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)

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

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

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

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

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)

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

Consider any schedule with any interleaving of operations of \( \tau_1 \) and \( \tau_2 \)
An Example of 2PL

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

\[ \tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(A), \text{Read}_{\tau_1}(B), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(B), \text{Release}_{\tau_1}(A); \]

\[ \tau_2 = \text{Lock}_{\tau_2}(E), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(A), \text{Write}_{\tau_2}(A), \text{Read}_{\tau_2}(E), \text{Write}_{\tau_2}(E), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(E). \]

Consider any schedule with any interleaving of operations of \( \tau_1 \) and \( \tau_2 \)

- If \( \tau_1 \) executes \( \text{Lock}_{\tau_1}(A) \) before \( \tau_2 \) executes \( \text{Lock}_{\tau_2}(A) \): all read and write operations of \( \tau_1 \) effectively happen before those of \( \tau_2 \).
An Example of 2PL

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

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

Consider any schedule with any interleaving of operations of \( \tau_1 \) and \( \tau_2 \)

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

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

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Lock}_{\tau_1}(A)</td>
</tr>
<tr>
<td>\text{Lock}_{\tau_1}(B)</td>
</tr>
<tr>
<td>\text{Lock}_{\tau_2}(B)</td>
</tr>
<tr>
<td>\text{Lock}_{\tau_2}(A)</td>
</tr>
</tbody>
</table>

Deadlocks are one of the issues arising from lock contention.
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

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_{\tau_1}(A)</td>
</tr>
<tr>
<td>Lock_{\tau_1}(B)</td>
</tr>
</tbody>
</table>

Deadlocks are one of the issues arising from *lock contention*. 
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 Ana, then Bo, then Celeste, then Dafni, then Elisa, ... .

Example
Consider the transaction $\tau = \text{"if Bo has$500, then move$200 from Bo to Ana"}$. Any schedule for $\tau$ needs to start with:

- Lock $\tau(\text{Ana})$,
- Lock $\tau(\text{Bo})$,
- ...

we even lock Ana if Bo does not have funds.
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 Ana, then Bo, then Celeste, then Dafni, then Elisa, ....
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 Ana, then Bo, then Celeste, then Dafni, then Elisa, ....

Example

Consider the transaction

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

Any schedule for \( \tau \) needs to start with:

\[ \text{Lock}_\tau(Ana), \text{Lock}_\tau(Bo), \ldots, \]

we even lock Ana if Bo does not have funds.
Dealing with Deadlocks: Optimistic Approach

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: Optimistic Approach

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* or *prevention*.
Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention
If a transaction tries to obtain a lock that is already held: \textit{abort the transaction entirely}.

- No need for \textit{deadlock detection} or \textit{prevention}.
- Very easy to implement.
Dealing with Deadlocks: Optimistic Approach

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* or *prevention*.
- Very easy to implement.
- Minimizes the costs for transactions that are able to commit.
Dealing with Deadlocks: Optimistic Approach

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* or *prevention*.
- 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.
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).

To improve performance, you can *give up* on serializability!
### Degrees of Isolation in SQL

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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](https://doi.org/10.1145/568271.223785) and [https://doi.org/10.1016/0950-5849(96)01109-3](https://doi.org/10.1016/0950-5849(96)01109-3) are recommended.
## Degrees of Isolation in SQL

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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.

---

1There 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\(^1\)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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.

\(^1\)There are excellent papers on this topic! E.g., [https://doi.org/10.1145/568271.223785](https://doi.org/10.1145/568271.223785) and [https://doi.org/10.1016/0950-5849(96)01109-3](https://doi.org/10.1016/0950-5849(96)01109-3) are recommended.
**Degrees of Isolation in SQL**

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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.

---

1There are excellent papers on this topic! E.g., [https://doi.org/10.1145/568271.223785](https://doi.org/10.1145/568271.223785) and [https://doi.org/10.1016/0950-5849(96)01109-3](https://doi.org/10.1016/0950-5849(96)01109-3) are recommended.
## Degrees of Isolation in SQL

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.

---

1There are excellent papers on this topic! E.g., [https://doi.org/10.1145/568271.223785](https://doi.org/10.1145/568271.223785) and [https://doi.org/10.1016/0950-5849(96)01109-3](https://doi.org/10.1016/0950-5849(96)01109-3) are recommended.
## Degrees of Isolation in SQL\(^1\)

<table>
<thead>
<tr>
<th>Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Read</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

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.

---

\(^1\)There are excellent papers on this topic! E.g., [https://doi.org/10.1145/568271.223785](https://doi.org/10.1145/568271.223785) and [https://doi.org/10.1016/0950-5849(96)01109-3](https://doi.org/10.1016/0950-5849(96)01109-3) are recommended.
Safe Execution without Locks

Concurrent transaction execution can make sense \textit{without} isolation. E.g., one can use application-specific knowledge!
Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation.
E.g., one can use application-specific knowledge!

A Banking System
Observe: undoing a withdraw increases balance, undoing deposits decreases balance!
Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation. E.g., one can use application-specific knowledge!

A Banking System

Observe: undoing a withdraw increases balance, undoing deposits decreases balance!

Consider executions in which all steps can:

- always withdraw money;
- only deposit money after either *commit* or *abort* is decided.
Safe Execution without Locks

Concurrent transaction execution can make sense without isolation. E.g., one can use application-specific knowledge!

A Banking System
Observe: undoing a withdraw increases balance, undoing deposits decreases balance!

Consider executions in which all steps can:
- always withdraw money;
- only deposit money after either commit or abort is decided.

These executions guarantee that no account will have a negative balance!
Ingredients of Sharding in a Resilient Environment

Multi-shard transaction execution of $\tau$ requires

**Replication** of $\tau$ among shards.
- E.g., a two-phase commit step.

**Concurrency control** to guarantee consistent execution of $\tau$.
- E.g., using *locks* to prevent concurrent access to accounts.

To One needs *computations* within a shard and *communication* between shards.
Ingredients of Sharding in a Resilient Environment

Multi-shard transaction execution of $\tau$ requires

- **Replication** of $\tau$ among shards.
  
  E.g., a two-phase commit step.

- **Concurrency control** to guarantee consistent execution of $\tau$.
  
  E.g., using *locks* to prevent concurrent access to accounts.

To One needs *computations* within a shard and *communication* between shards.

**Fault-tolerant shards**

Each shard is a cluster of replicas that can be faulty.

- **Consensus** for each *computation* within shards.

- **Cluster-sending** for any *communication* between shards.

Consensus is costly: Minimize its use.
Consider a multi-shard transaction $\tau$:

- Processing is broken down into three types of *shard-steps*: vote, commit, and abort.
- Each shard-step is performed via *one* consensus step.
- Transfer control between steps using *cluster-sending*.

**Execution method** determines the local operations of a shard-step:

- *locks*, *checking conditions*, *updating state*, ... .

**Orchestration method** determines how *control is transferred* between shard-steps:
- perform *votes*, collect *votes*, decide *commit* or *abort* $\tau$. 
Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

\[ \tau = \text{“if } Ana \text{ has $500 and } Bo \text{ has $200, then move $400 from } Ana \text{ to } Bo.” } \]
Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

\[ \tau = \text{“if } Ana \text{ has }$500 \text{ and } Bo \text{ has }$200, \text{ then move }$400 \text{ from } Ana \text{ to } Bo.” \]

\[ \sigma_1 = \text{“Lock}_\tau(Ana); \text{ if } Ana \text{ has }$500, \text{ then forward } \sigma_2 \text{ to } C_b \text{ (commit vote) else Release}_\tau(Ana) \text{ (abort vote).”} \]

\textbf{vote-step}

\[ \sigma_1 \text{ at } C_a \]
Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

\[ \tau = \text{“if } \text{Ana has } \$500 \text{ and Bo has } \$200, \text{ then move } \$400 \text{ from Ana to Bo.”} \]

\[ \sigma_2 = \text{“Lock}_\tau(\text{Bo}); \text{ if Bo has } \$200, \text{ then add } \$400 \text{ to Bo; Release}_\tau(\text{Bo}); \text{ and forward } \sigma_3 \text{ to } C_a \text{ (commit)} \]

\[ \text{else Release}_\tau(\text{Bo}) \text{ and forward } \sigma_4 \text{ to } C_a \text{ (abort).”} \]

vote-step  vote-step

\[ \sigma_1 \text{ at } C_a \overset{\text{vote commit}}{\longrightarrow} \sigma_2 \text{ at } C_b \]
Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

\[ \tau = \text{“if Ana has $500 and Bo has $200, then move $400 from Ana to Bo.”} \]

\[ \sigma_3 = \text{“remove $400 from Ana and Release}_\tau(Ana).” \]

\[ \sigma_4 = \text{“Release}_\tau(Ana).” \]
Resilient Orchestration Methods

Orchestration \(\approx\) two-phase commit, except that *shards never fail*.

Vote-steps in *sequence*, decide *centralized*, commit or abort in *parallel*. 
Resilient Orchestration Methods

Orchestration ≈ two-phase commit, except that *shards never fail.*

Vote-steps in *parallel*, decide *centralized*, commit or abort in *parallel.*
Resilient Orchestration Methods

Orchestration ≈ two-phase commit, except that *shards never fail*.

Vote-steps in *parallel*, decide *decentralized*, commit or abort in *parallel*. 
Resilient Execution Methods

Execution updates state and performs *concurrency control*.

- Write uncommitted execution for *free*:
  Due to consensus, shard-steps are performed in sequence on that shard.

- Higher isolation levels via *two-phase locking*:
  - read uncommitted execution: only *write locks*;
  - read committed execution: *read locks* during steps;
  - serializable execution: *read and write locks*.

- Blocking locks (with linear orchestration) versus non-blocking locks.
Evaluation

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
<th>Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsafe</strong></td>
<td><strong>safe</strong></td>
<td><strong>Read Uncommitted</strong></td>
</tr>
<tr>
<td><strong>blocking</strong></td>
<td><strong>non-blocking</strong></td>
<td><strong>blocking</strong></td>
</tr>
<tr>
<td>Linear</td>
<td>- LIFu</td>
<td>- LIFs</td>
</tr>
<tr>
<td>Centralized</td>
<td>- CIFu</td>
<td>- CIFs</td>
</tr>
<tr>
<td>Distributed</td>
<td>- DIFu</td>
<td>- DIFs</td>
</tr>
</tbody>
</table>

- **Total Runtime**
- **Average Committed Throughput**
### Evaluation

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe</td>
<td>safe</td>
<td>blocking</td>
<td>non-blocking</td>
<td>blocking</td>
<td>non-blocking</td>
</tr>
<tr>
<td>Linear</td>
<td>LI Fu</td>
<td>LI Fs</td>
<td>LR Ub</td>
<td>LR Unb</td>
<td>LR Cb</td>
</tr>
<tr>
<td>Centralized</td>
<td>CI Fu</td>
<td>CI Fs</td>
<td>CR Unb</td>
<td>CR Cnb</td>
<td>CS nb</td>
</tr>
<tr>
<td>Distributed</td>
<td>DI Fu</td>
<td>DI Fs</td>
<td>DR Unb</td>
<td>DR Cnb</td>
<td>DS nb</td>
</tr>
</tbody>
</table>

**Total Runtime**

**Cumulative Duration**
**Evaluation**

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
<th>Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsafe</strong></td>
<td><strong>safe</strong></td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>Read Uncommitted</td>
<td>Read Committed</td>
</tr>
<tr>
<td>Centralized</td>
<td><strong>blocking</strong></td>
<td><strong>non-blocking</strong></td>
</tr>
<tr>
<td>Distributed</td>
<td><strong>blocking</strong></td>
<td><strong>non-blocking</strong></td>
</tr>
<tr>
<td>Linear</td>
<td>LIFu</td>
<td>LIFs</td>
</tr>
<tr>
<td>Centralized</td>
<td>CIFu</td>
<td>CIFs</td>
</tr>
<tr>
<td>Distributed</td>
<td>DIFu</td>
<td>DIFs</td>
</tr>
</tbody>
</table>

### Average Throughput

-10^4

-10^4

Throughput (txn/s)

Number of Shards

Throughput (txn/s)

Number of Shards
**Evaluation**

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe</td>
<td>safe</td>
</tr>
<tr>
<td>Linear</td>
<td>LIFu</td>
</tr>
<tr>
<td>Centralized</td>
<td>CIFu</td>
</tr>
<tr>
<td>Distributed</td>
<td>DIFu</td>
</tr>
</tbody>
</table>

**Step per Shard**

<table>
<thead>
<tr>
<th>Steps per Shard</th>
<th>Number of Shards</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>$2^0$</td>
</tr>
<tr>
<td>0.1</td>
<td>$2^1$</td>
</tr>
<tr>
<td>0.2</td>
<td>$2^2$</td>
</tr>
<tr>
<td>0.3</td>
<td>$2^3$</td>
</tr>
<tr>
<td>0.4</td>
<td>$2^4$</td>
</tr>
<tr>
<td>0.5</td>
<td>$2^5$</td>
</tr>
<tr>
<td>0.6</td>
<td>$2^6$</td>
</tr>
<tr>
<td>0.7</td>
<td>$2^7$</td>
</tr>
<tr>
<td>0.8</td>
<td>$2^8$</td>
</tr>
<tr>
<td>0.9</td>
<td>$2^9$</td>
</tr>
</tbody>
</table>

**Median Consensus Steps**

- $10^4$
- $10^3$

**Shard-Step Imbalance**

- $10^4$
- $10^3$

**Runtime (s)**

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

**Throughput (txn/Steps per Shard)**

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

**Number of Shards**

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
## Evaluation

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
<th>Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsafe</strong></td>
<td><strong>safe</strong></td>
<td>Read Uncommitted</td>
</tr>
<tr>
<td>Linear</td>
<td>- LIFu</td>
<td>- LRUb</td>
</tr>
<tr>
<td>Centralized</td>
<td>- CIFu</td>
<td>- CRUb</td>
</tr>
<tr>
<td>Distributed</td>
<td>- DIFu</td>
<td>- DRUb</td>
</tr>
</tbody>
</table>

### Median Consensus Steps

![Median Consensus Steps Graph]

### Shard-Step Imbalance

![Shard-Step Imbalance Graph]
### Evaluation

<table>
<thead>
<tr>
<th>Isolation-Free execution (write uncommitted)</th>
<th>Lock-based execution</th>
<th>Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsafe</strong></td>
<td><strong>blocking</strong></td>
<td><strong>blocking</strong></td>
</tr>
<tr>
<td>Linear</td>
<td>LIFu</td>
<td>LRCb</td>
</tr>
<tr>
<td>Centralized</td>
<td>CIFu</td>
<td>CRCnb</td>
</tr>
<tr>
<td>Distributed</td>
<td>DIFu</td>
<td>DRCnb</td>
</tr>
<tr>
<td><strong>safe</strong></td>
<td><strong>non-blocking</strong></td>
<td><strong>non-blocking</strong></td>
</tr>
<tr>
<td></td>
<td>LIFs</td>
<td>LRCnb</td>
</tr>
<tr>
<td></td>
<td>CIFs</td>
<td>CRCnb</td>
</tr>
<tr>
<td></td>
<td>DIFs</td>
<td>DRCnb</td>
</tr>
</tbody>
</table>

#### Constraint Failures

![Constraint Failures Graph]

#### Failed Locks

![Failed Locks Graph]