Transactions

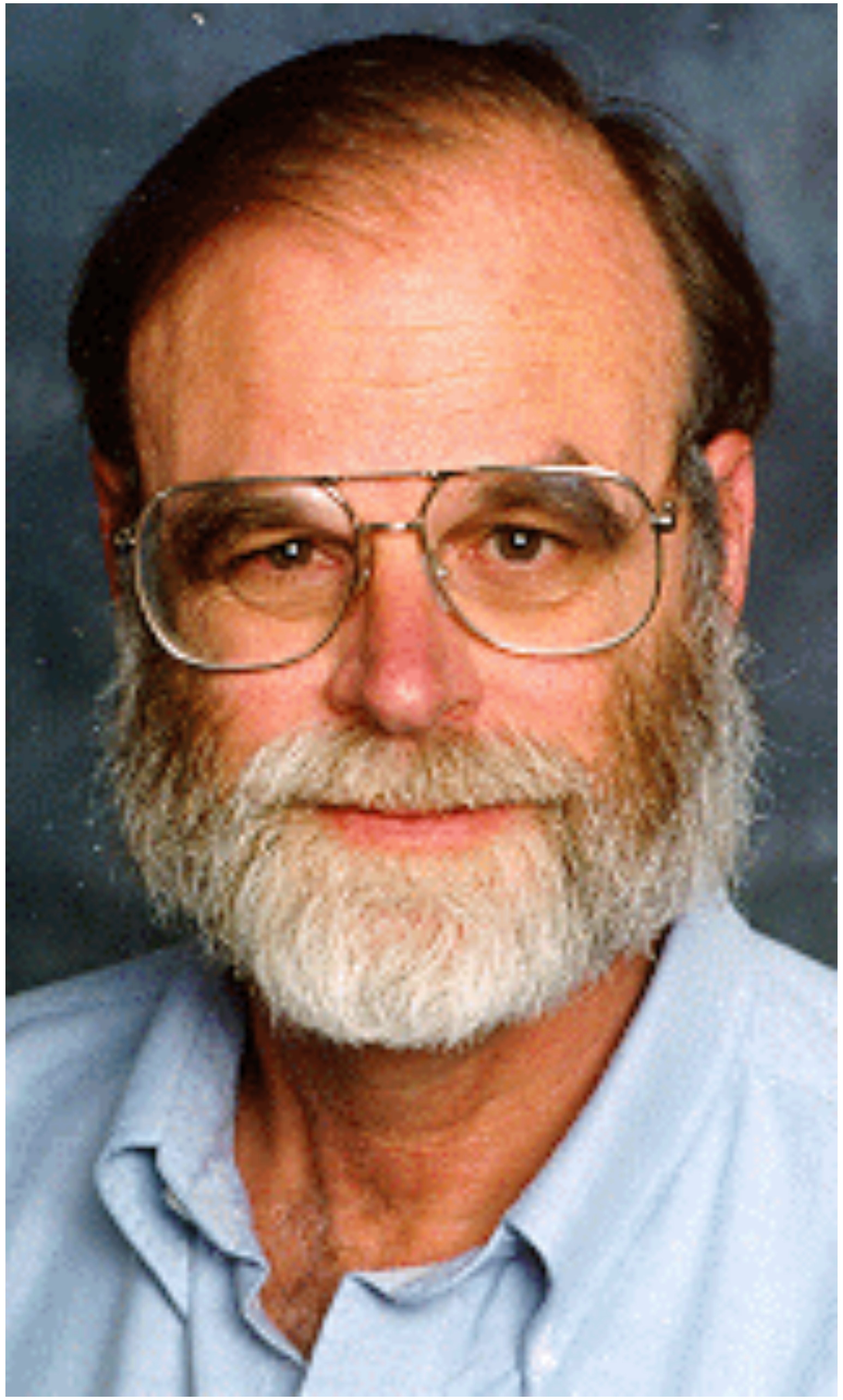
Prof. George Candea
School of Computer & Communication Sciences

What is a transaction in the real world ?

- Two or more parties…
  - *negotiate for a while*
  - *then make a deal*
  - *write it up in a contract*
  - *all parties sign the contract*
    => *transaction completes*
- Implication
  - *everyone agrees*
  - *deal is binding*

Properties of real-world transactions

- Transaction is in accordance with legal protocols
  - *i.e., law governs society*
- The entire deal either takes place or not
  - *either all parties are bound by it or none are*
- Once the contract is signed, it cannot be abrogated
  - *can be amended / compensated*
- If someone engages in a different transaction doesn't affect this one



Tandem TR 81.3

The Transaction Concept: Virtues and Limitations

Jim Gray
Tandem Computers Incorporated
19333 Vallco Parkway, Cupertino CA 95014

June 1981

ABSTRACT: A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation). The transaction concept is key to the structuring of data management applications. The concept may have applicability to programming systems in general. This paper restates the transaction concepts and attempts to put several implementation approaches in perspective. It then describes some areas which require further study: (1) the integration of the transaction concept with the notion of abstract data type, (2) some techniques to allow transactions to be composed of sub-transactions, and (3) handling transactions which last for extremely long times (days or months).

Appeared in Proceedings of Seventh International Conference on Very Large Databases, Sept. 1981. Published by Tandem Computers Incorporated.

What is a transaction in the computing world ?

- Transaction = collection of *actions* that comprise a consistent transformation of system state
  - *Actions read and transform values*
- Outcome = committed | aborted
- The only way to "correct" a committed transaction is via another (compensating) transaction
- System state may include assertions of what consistency means

What is an action in a transaction ?

- Unprotected
  - *need not be undone if txn must be aborted*
  - *need not be redone if the value needs to be reconstructed*
- Protected
  - *action can and must be undone / redone if …*
- Real
  - *cannot be undone (once done)*

- Txn commits => all protected and real actions persist

  Txn aborts => no effects of protected and real actions are visible to other txns

How does a transaction look ?

```
DELETE FROM Orders WHERE ClientID = @DonaldTrump
```

How does a transaction look ?

```
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
```

How does a transaction look ?

```
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
```

```
BEGIN TRANSACTION
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
COMMIT
```

How does a transaction look ?

```
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
```

```
BEGIN TRANSACTION
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
COMMIT
```

```
BEGIN TRANSACTION
DELETE FROM Orders WHERE ClientID = @DonaldTrump
DELETE FROM Clients WHERE ClientID = @DonaldTrump
IF @@ROWCOUNT > 1
    ROLLBACK
COMMIT
```

```python
import sqlite3

conn = sqlite3.connect('database_file.db')
cursor = conn.cursor()
client_id = 'DonaldTrump'

try:
    cursor.execute("DELETE FROM Orders WHERE ClientID = ?", (client_id,))
    cursor.execute("DELETE FROM Clients WHERE ClientID = ?", (client_id,))

    if cursor.rowcount > 1:
        conn.rollback()  # Oops, more than one client was removed => abort
    else:
        conn.commit()    # All's good, commit the transaction

except Exception as e:
   conn.rollback()    # Something went wrong -> abort

finally:
    cursor.close()
    conn.close()
```

```python
import sqlite3

conn = sqlite3.connect('database_file.db')
cursor = conn.cursor()
client_id = 'DonaldTrump'

try:
    cursor.execute("DELETE FROM Orders WHERE ClientID = ?", (client_id,))
    cursor.execute("DELETE FROM Clients WHERE ClientID = ?", (client_id,))

    if cursor.rowcount > 1:
        conn.rollback()  # Oops, more than one client was removed => abort
    else:
        conn.commit()    # All's good, commit the transaction

except Exception as e:
   conn.rollback()    # Something went wrong -> abort

finally:
    cursor.close()
    conn.close()
```

```python
import sqlite3

conn = sqlite3.connect('database_file.db')
cursor = conn.cursor()
client_id = 'DonaldTrump'

try:
    cursor.execute("DELETE FROM Orders WHERE ClientID = ?", (client_id,))
    cursor.execute("DELETE FROM Clients WHERE ClientID = ?", (client_id,))

    if cursor.rowcount > 1:
        conn.rollback()  # Oops, more than one client was removed => abort
    else:
        conn.commit()    # All's good, commit the transaction

except Exception as e:
   conn.rollback()    # Something went wrong -> abort

finally:
    cursor.close()
    conn.close()
```

```python
import sqlite3

conn = sqlite3.connect('database_file.db')
cursor = conn.cursor()
client_id = 'DonaldTrump'

try:
    cursor.execute("DELETE FROM Orders WHERE ClientID = ?", (client_id,))
    cursor.execute("DELETE FROM Clients WHERE ClientID = ?", (client_id,))

    if cursor.rowcount > 1:
        conn.rollback()  # Oops, more than one client was removed => abort
    else:
        conn.commit()    # All's good, commit the transaction

except Exception as e:
   conn.rollback()    # Something went wrong -> abort

finally:
    cursor.close()
    conn.close()
```

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from my_models import Order, Client

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)

client_id = 'DonaldTrump'

try:
   with Session() as session:
       orders_deleted = session.query(Order).filter(Order.client_id == client_id).delete()
       clients_deleted = session.query(Client).filter(Client.client_id == client_id).delete()

       if clients_deleted > 1:
           session.rollback()  # Oops, more than one client was removed => abort
       else:
           session.commit()    # All's good, commit the transaction

except Exception as e:
   session.rollback()      # Something went wrong -> abort
```

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from my_models import Order, Client

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)

client_id = 'DonaldTrump'

try:
   with Session() as session:
       orders_deleted = session.query(Order).filter(Order.client_id == client_id).delete()
       clients_deleted = session.query(Client).filter(Client.client_id == client_id).delete()

       if clients_deleted > 1:
           session.rollback()  # Oops, more than one client was removed => abort
       else:
           session.commit()    # All's good, commit the transaction

except Exception as e:
   session.rollback()      # Something went wrong -> abort
```

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from my_models import Order, Client

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)

client_id = 'DonaldTrump'

try:
   with Session() as session:
       orders_deleted = session.query(Order).filter(Order.client_id == client_id).delete()
       clients_deleted = session.query(Client).filter(Client.client_id == client_id).delete()

       if clients_deleted > 1:
           session.rollback()  # Oops, more than one client was removed => abort
       else:
           session.commit()    # All's good, commit the transaction

except Exception as e:
   session.rollback()      # Something went wrong -> abort
```

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from my_models import Order, Client

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)

client_id = 'DonaldTrump'

try:
    with Session() as session:
        orders_deleted = session.query(Order).filter(Order.client_id == client_id).delete()
        clients_deleted = session.query(Client).filter(Client.client_id == client_id).delete()

        if clients_deleted > 1:
            session.rollback()  # Oops, more than one client was removed => abort
        else:
            session.commit()    # All's good, commit the transaction

except Exception as e:
    session.rollback()      # Something went wrong -> abort
```

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from my_models import Order, Client

engine = create_engine('sqlite:///example.db')
Session = sessionmaker(bind=engine)

client_id = 'DonaldTrump'

try:
   with Session() as session:
       orders_deleted = session.query(Order).filter(Order.client_id == client_id).delete()
       clients_deleted = session.query(Client).filter(Client.client_id == client_id).delete()

       if clients_deleted > 1:
           session.rollback()  # Oops, more than one client was removed => abort
       else:
           session.commit()    # All's good, commit the transaction

except Exception as e:
   session.rollback()      # Something went wrong -> abort
```

ACID

A = Atomicity "All or nothing"

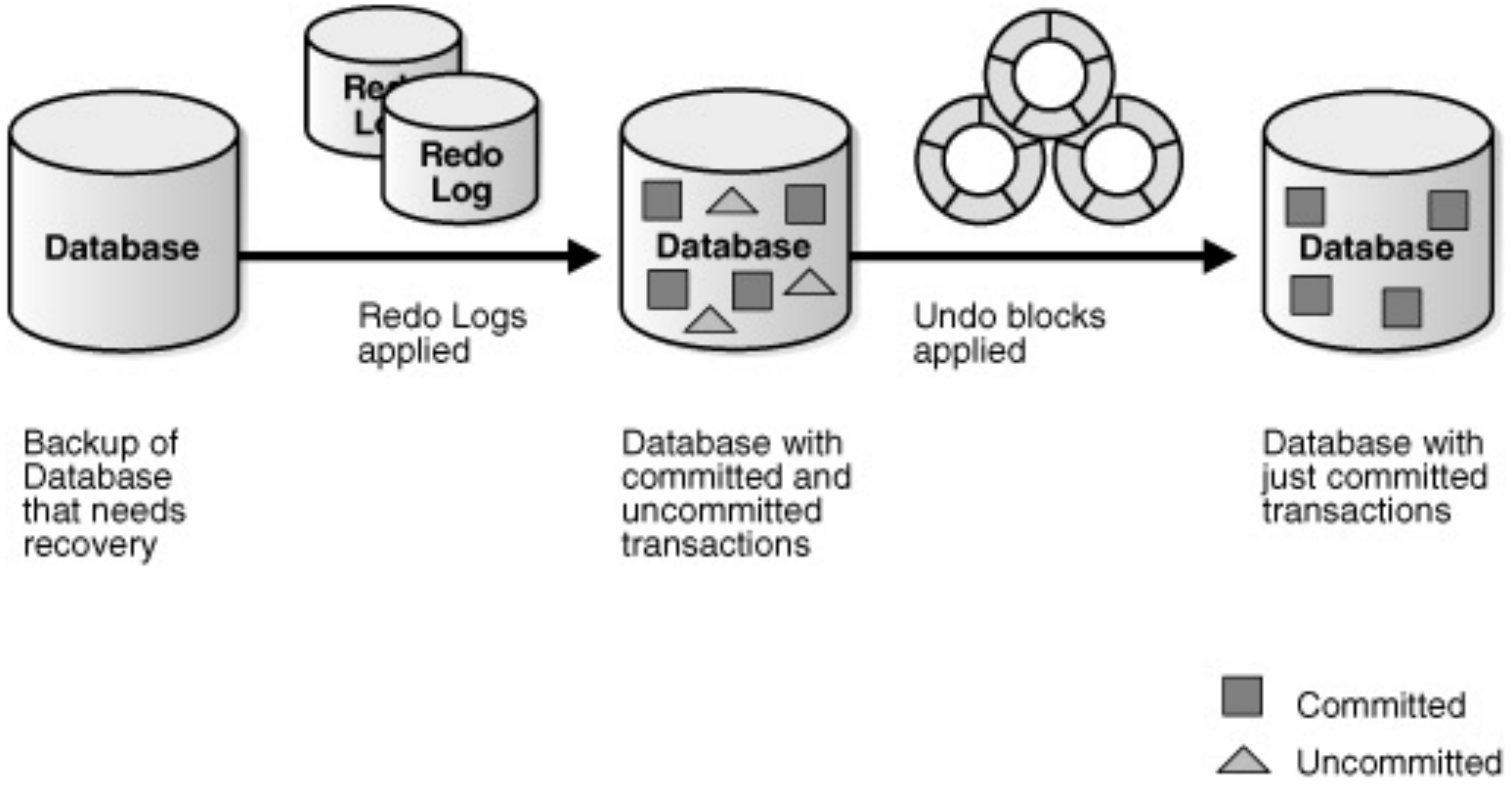
- Either all protected and real actions are visible or none

A = Atomicity "All or nothing"

- Either all protected and real actions are visible or none
- Key = how txn looks "from the outside"
  - *expressed in terms of abstract state*
  - *partial results ok, as long as not visible*

A = Atomicity "All or nothing"

- Either all protected and real actions are visible or none
- Key = how txn looks "from the outside"
  - *expressed in terms of abstract state*
  - *partial results ok, as long as not visible*



https://blog.mihai.tech/2019/oracle-archive-flash-logs-redo-logs-undo-logs/

C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

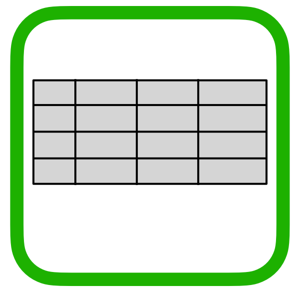
Object O

C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

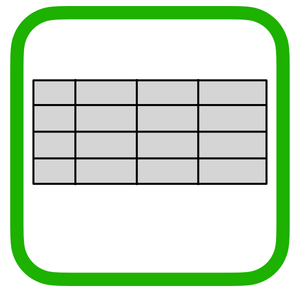
Abstraction O.abs

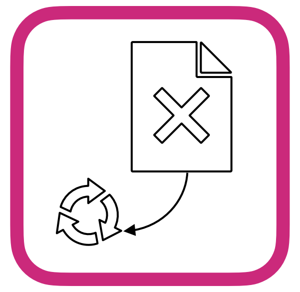


C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

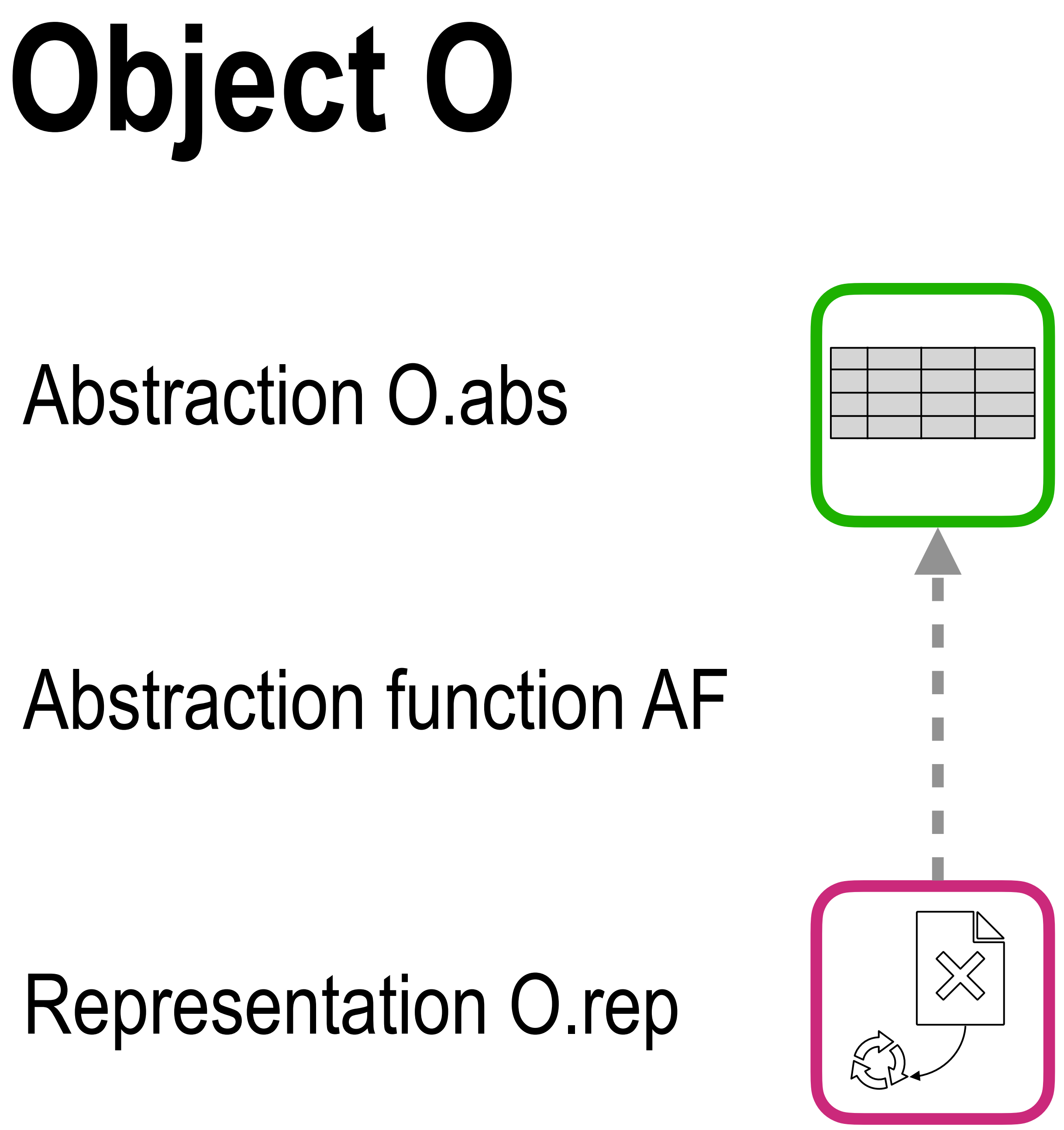
Object O

Abstraction O.abs 

Representation O.rep 

C = Consistency
"Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*



C = Consistency — "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction

$\Uparrow$

O.abs = AF(O.rep)

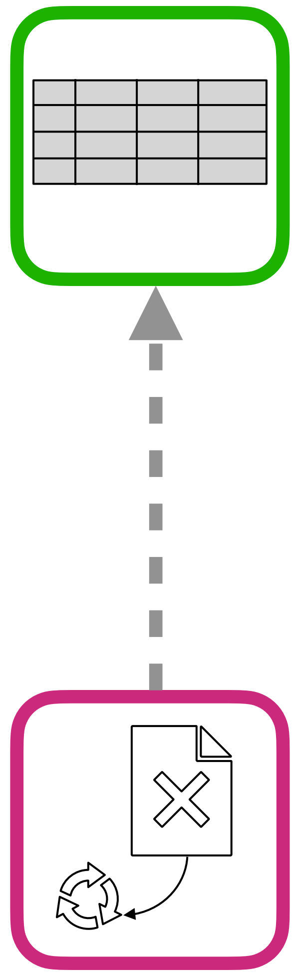
$\Uparrow$

RI(O.rep)=true

Abstraction O.abs

Abstraction function AF

Representation O.rep



C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction

$\Uparrow$

O.abs = AF(O.rep)

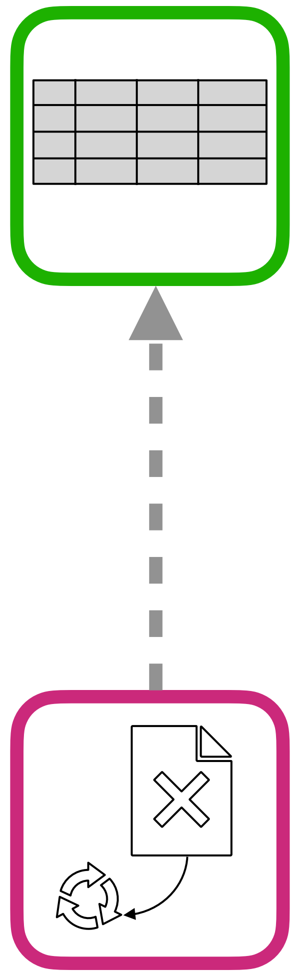
$\Uparrow$

$\boxed{\text{RI(O.rep)=true}}$

Abstraction O.abs

Abstraction function AF

Representation O.rep



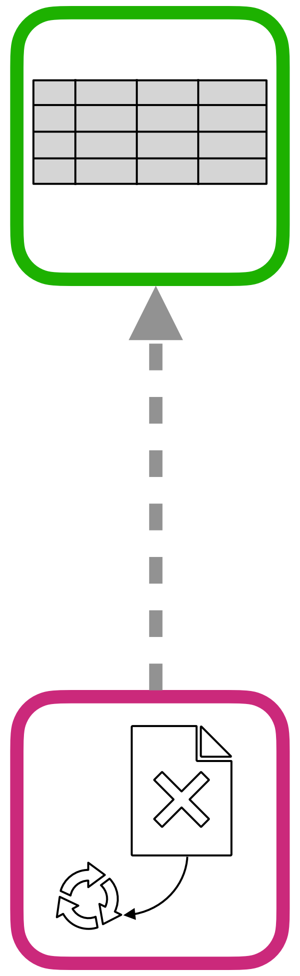
C = Consistency

“Obey legal protocols”

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction	Abstraction O.abs
$\Uparrow$	
O.abs = AF(O.rep)	Abstraction function AF
$\Uparrow$	
RI(O.rep)=true	Representation O.rep



C = Consistency

“Obey legal protocols”

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction

⇑

O.abs = AF(O.rep)

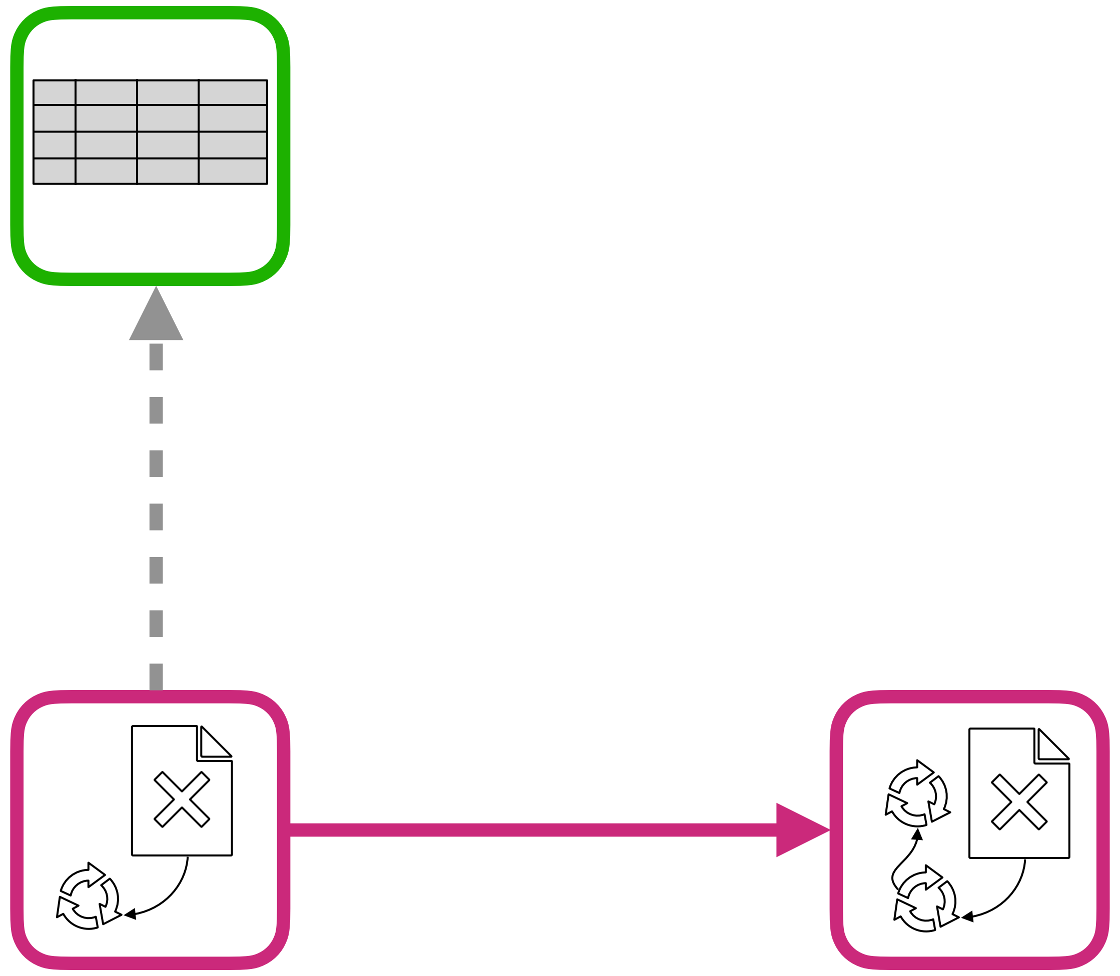
⇑

RI(O.rep)=true

Abstraction O.abs

Abstraction function AF

Representation O.rep



C = Consistency "Obey legal protocols"

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction

⇑

O.abs = AF(O.rep)

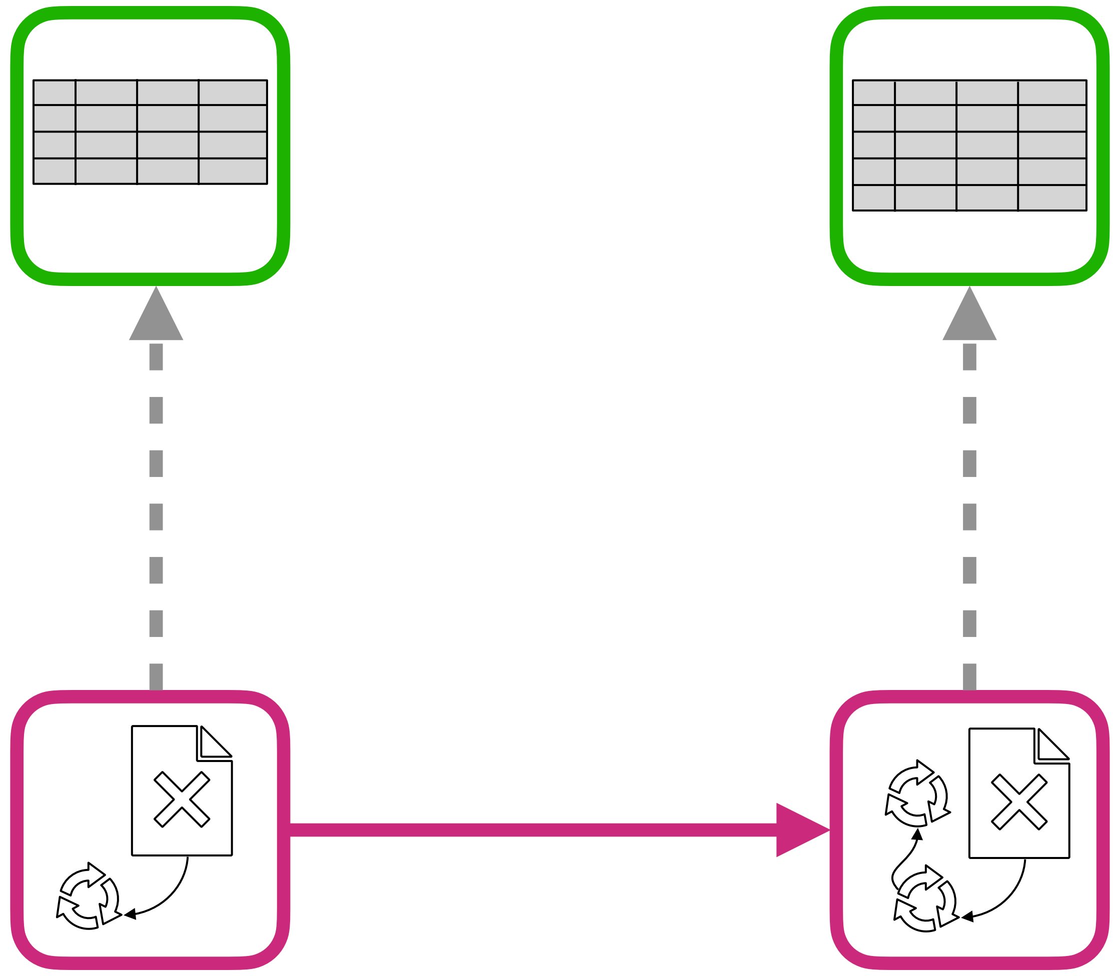
⇑

RI(O.rep)=true

Abstraction O.abs

Abstraction function AF

Representation O.rep



C = Consistency

“Obey legal protocols”

- Txn transitions system from one valid state to another
  - *intermediate states are not visible*

Object O

O exposes correct abstraction

$\Uparrow$

O.abs = AF(O.rep)

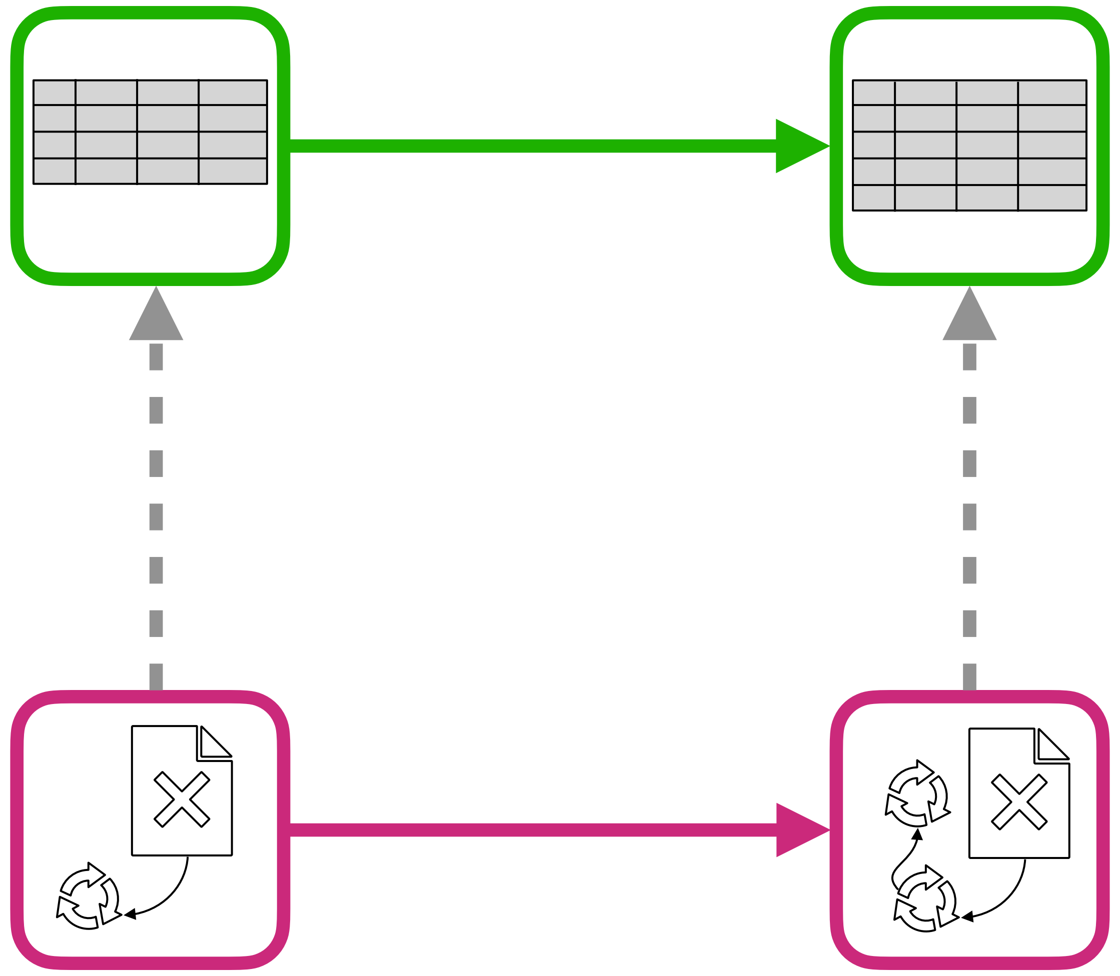
$\Uparrow$

RI(O.rep)=true

Abstraction O.abs

Abstraction function AF

Representation O.rep



Integrity Constraints

```
CREATE TABLE Clients(
   Id int NOT NULL PRIMARY KEY,
   ...
)
```

Integrity Constraints

```
CREATE TABLE Clients(
   Id int NOT NULL PRIMARY KEY,
   ...
)

CREATE TABLE Orders (
   OrderId int NOT NULL PRIMARY KEY,
   ...
   ClientId int FOREIGN KEY REFERENCES Clients(Id)
)
```

Integrity Constraints

```
CREATE TABLE Clients(
   Id int NOT NULL PRIMARY KEY,
   ...
)
```

```
CREATE TABLE Orders (
   OrderId int NOT NULL PRIMARY KEY,
   ...
   ClientId int FOREIGN KEY REFERENCES Clients(Id)
)
```

```
CREATE TABLE Orders (
   OrderId int NOT NULL PRIMARY KEY,
   ...
   ClientId int FOREIGN KEY REFERENCES Clients(Id) ON DELETE CASCADE
)
```

C = Consistency "Obey legal protocols"

- guarantee is simply as strong as the defined rules
  - *If application-level code translates all its semantics into such constraints, then an ACID system guarantees application-level consistency*
- Is a txn-level property, restricting what the transaction itself can do

D = Durability

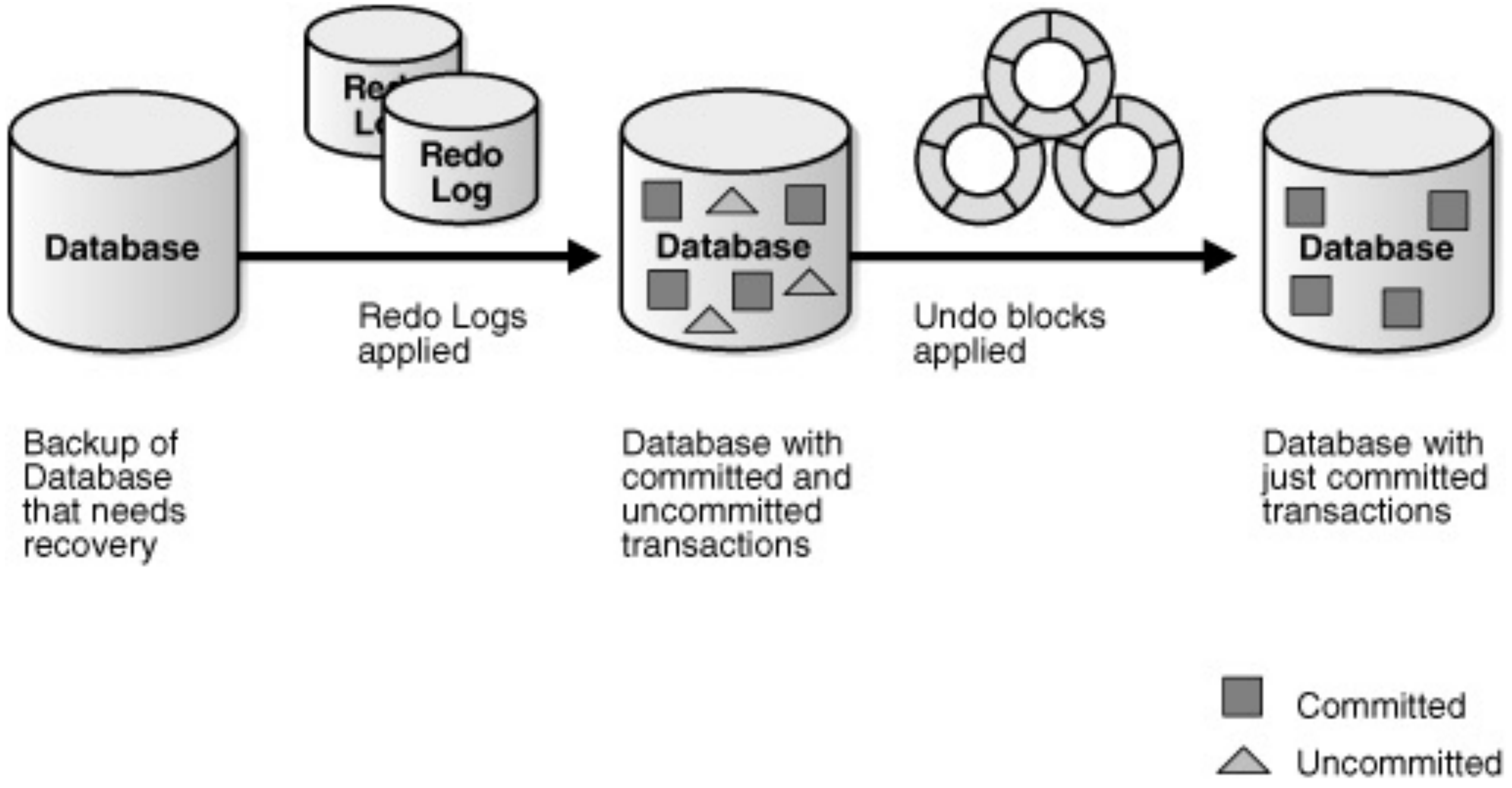
“Data is forever”

- A committed transaction cannot be undone by any failure

D = Durability

“Data is forever”

- A committed transaction cannot be undone by any failure
- What is the price of accomplishing this?

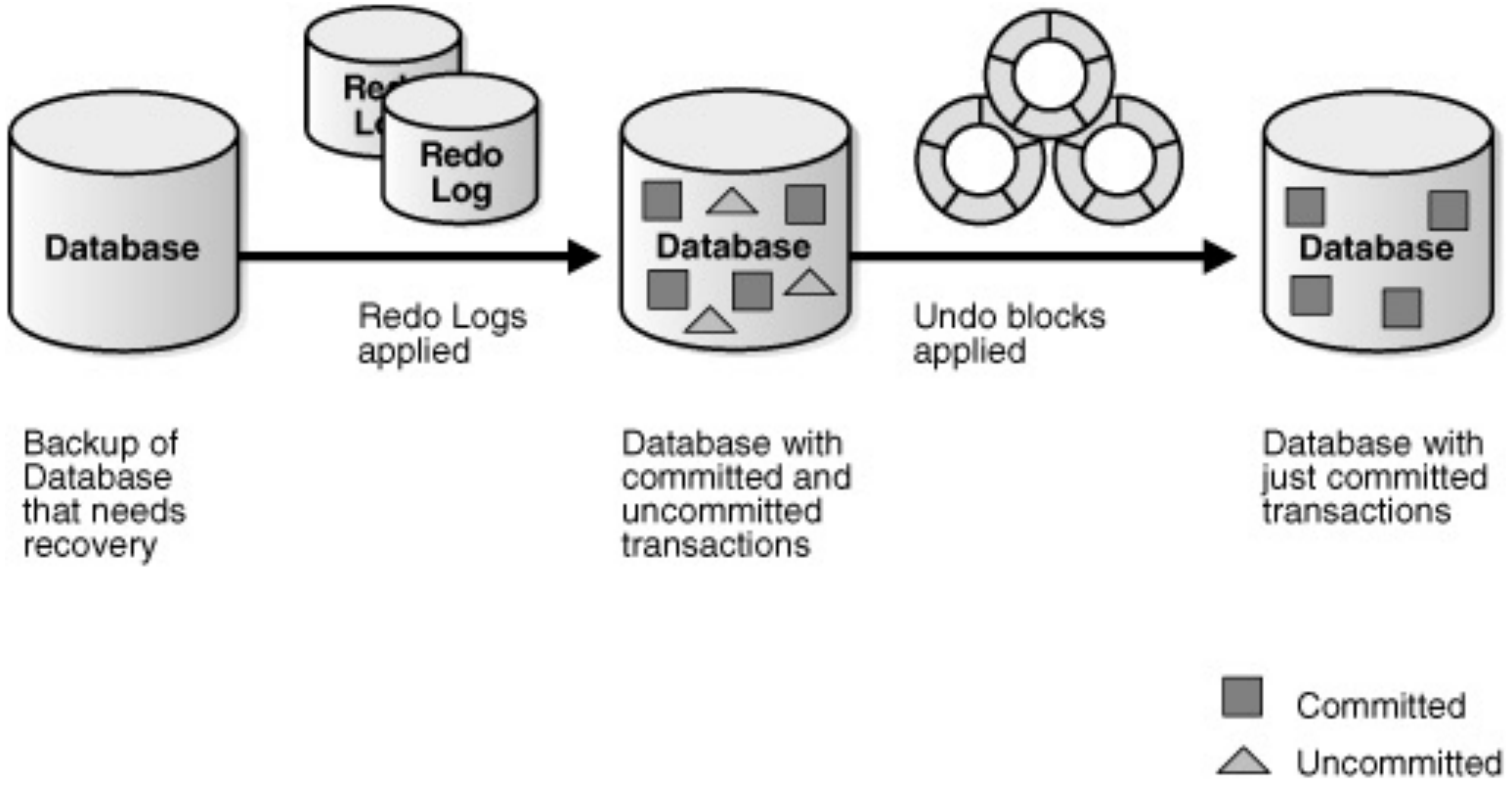


https://blog.mihai.tech/2019/oracle-archive-flash-logs-redo-logs-undo-logs/

D = Durability

"Data is forever"

- A committed transaction cannot be undone by any failure
- What is the price of accomplishing this?
- How do you choose how much to do/pay?



https://blog.mihai.tech/2019/oracle-archive-flash-logs-redo-logs-undo-logs/

I = Isolation "Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*

I = Isolation "Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*
- Strict isolation:
  - *Txn T has inputs I and outputs O*
  - *Other txns can read I but cannot read or write O before T commits*

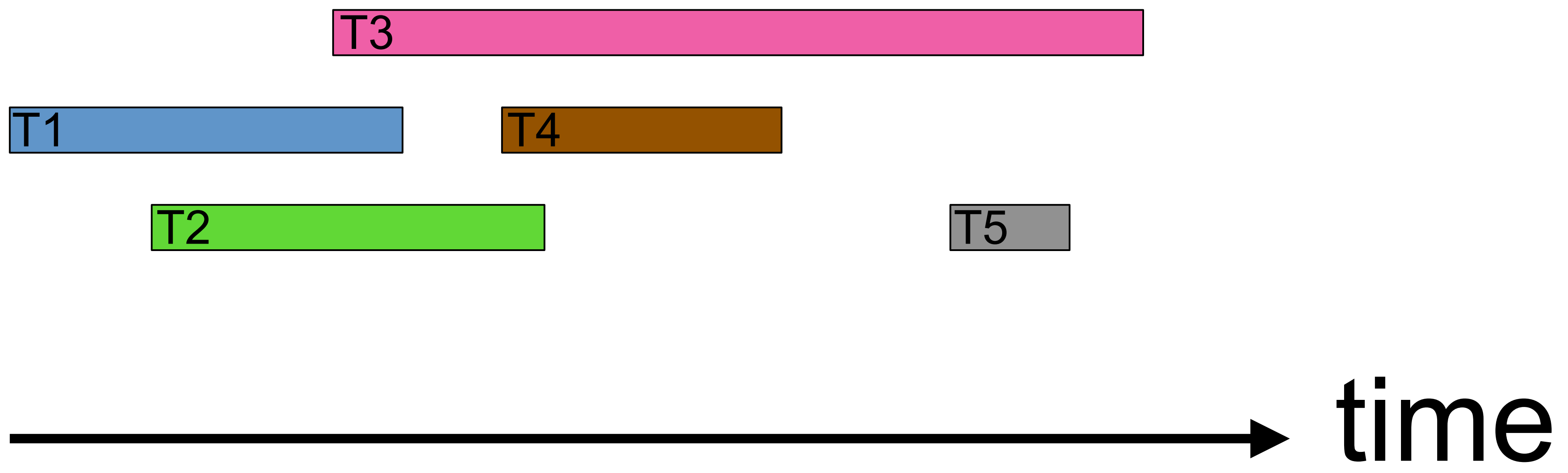
I = Isolation "Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*
- Strict isolation:
  - *Txn T has inputs I and outputs O*
  - *Other txns can read I but cannot read or write O before T commits*
- Serializable execution & serialization points

I = Isolation

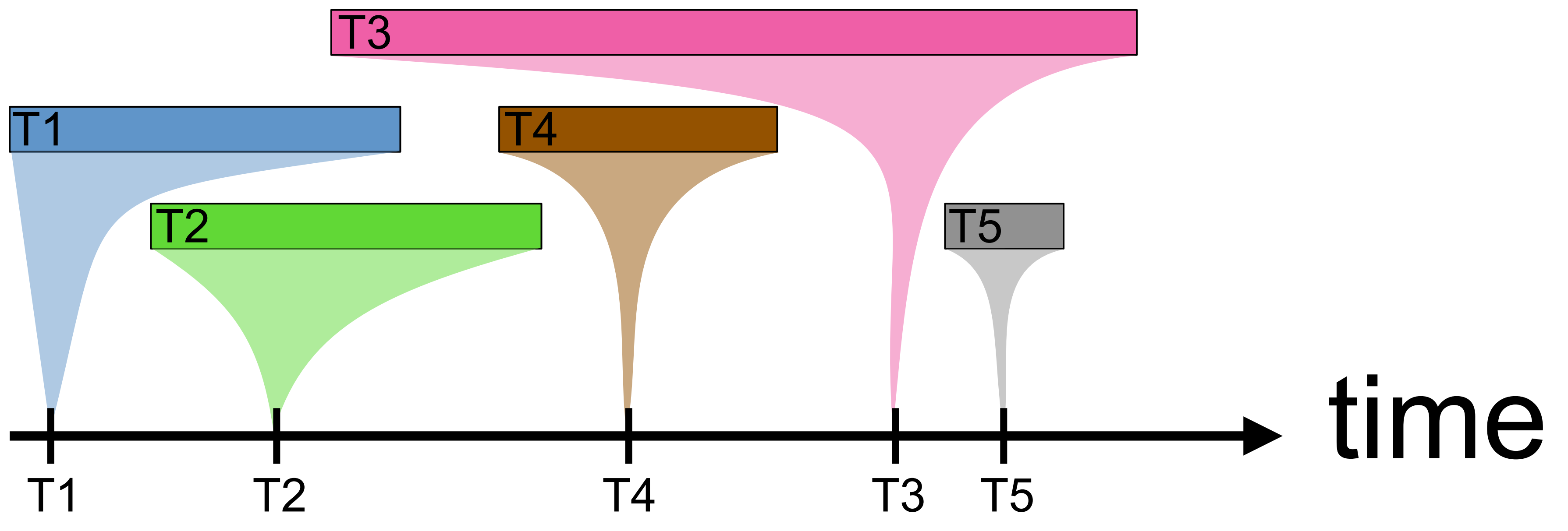
"Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*
- Strict isolation:
  - *Txn T has inputs I and outputs O*
  - *Other txns can read I but cannot read or write O before T commits*
- Serializable execution & serialization points



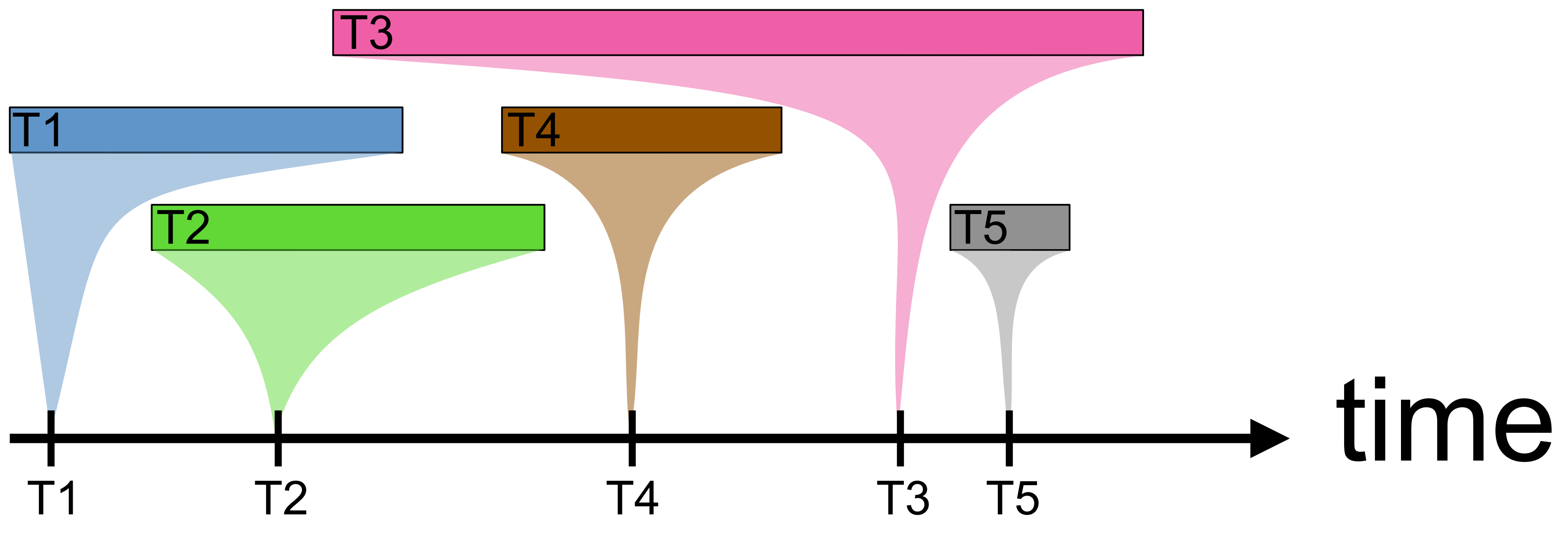
I = Isolation "Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*
- Strict isolation:
  - *Txn T has inputs I and outputs O*
  - *Other txns can read I but cannot read or write O before T commits*
- Serializable execution & serialization points



I = Isolation "Each transaction runs alone"

- Txns run concurrently —> it's as if each one runs on its own
  - *each txn commits before a new one starts*
- Strict isolation:
  - *Txn T has inputs I and outputs O*
  - *Other txns can read I but cannot read or write O before T commits*
- Serializable execution & serialization points
- Can sacrifice serializability for performance
  - *Hard to do ACID at scale*
  - *Introduces complexity in applications*



A = Atomicity

C = Consistency

I = Isolation

D = Durability

Nested Transactions

Nested transactions

- Customer calls the travel agent giving destination and travel dates.
  Agent negotiates with airlines for flights.
  Agent negotiates with car rental companies for cars.
  Agent negotiates with hotels for rooms.
  Agent receives tickets and reservations.
  Agent gives customer tickets and gets credit card number.
  Agent bills credit card.
  Customer uses tickets.

Nested transactions

- Customer calls the travel agent giving destination and travel dates.
  Agent negotiates with airlines for flights.
  Agent negotiates with car rental companies for cars.
  Agent negotiates with hotels for rooms.
  Agent receives tickets and reservations.
  Agent gives customer tickets and gets credit card number.
  Agent bills credit card.
  Customer uses tickets.
- Each step is a transaction and an action at the same time

Redefining the transaction

- Transaction = collection of
  - *Unprotected actions (don't require redo/undo)*
  - *Protected actions (need to be undoable/redoable)*
  - *Real actions (may be deferred but not undone)*
  - *Nested transactions which may be undone by invoking compensating transactions*

Redefining the transaction

- Transaction = collection of
  - *Unprotected actions (don't require redo/undo)*
  - *Protected actions (need to be undoable/redoable)*
  - *Real actions (may be deferred but not undone)*
  - *Nested transactions which may be undone by invoking compensating transactions*
- Nested txns != protected actions
  - *effects are visible to the outside world prior to the commit of the parent transaction*

Redefining the transaction

- Transaction = collection of
  - *Unprotected actions (don't require redo/undo)*
  - *Protected actions (need to be undoable/redoable)*
  - *Real actions (may be deferred but not undone)*
  - *Nested transactions which may be undone by invoking compensating transactions*
- Nested txns != protected actions
  - *effects are visible to the outside world prior to the commit of the parent transaction*
- Nested txn returns the name and params of the compensating txn
  - *keep in log of the parent txn*

Transactional Memory (TM)

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        sharedResource++;

        scope.Complete();                                    # commit txn
    }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                      # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Overview

- concurrency control mechanism

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                     # commit txn
    }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Overview

- concurrency control mechanism
- provide ACI but no D

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                      # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Overview

- concurrency control mechanism
- provide ACI but no D
- can be implemented in HW or SW

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                       # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

Software TM

- Library / programming language runtime
- Track read/write sets
- Optimistic concurrency control — check for conflicts at commit time
- Keep old versions until commit/abort

Hardware TM: Intel Transactional Synchronization Extensions

- Available in Intel's Skylake and ARM
- RTM = Restricted Transactional Memory
- Three new instructions:
  XBEGIN = start txnal execution
  XEND = end txnal execution
  XABORT = abort txnal execution

Intel TSX: XBEGIN and XEND

- Operand provides a relative offset to the fallback instruction address
  - *If the RTM region could not be successfully executed transactionally, jumps there*
  - *Post-abort, architectural state corresponds to that just before XBEGIN (`eax` contains abort status)*

Intel TSX: XBEGIN and XEND

- Operand provides a relative offset to the fallback instruction address
  - *If the RTM region could not be successfully executed transactionally, jumps there*
  - *Post-abort, architectural state corresponds to that just before XBEGIN (`eax` contains abort status)*
- XBEGIN instruction does not have fencing semantics
  - *but, upon abort, all memory updates inside RTM region are invisible*
  - *same semantics as LOCK-prefixed instructions but without the cost*

Intel TSX: XBEGIN and XEND

- Operand provides a relative offset to the fallback instruction address
  - *If the RTM region could not be successfully executed transactionally, jumps there*
  - *Post-abort, architectural state corresponds to that just before XBEGIN (`eax` contains abort status)*
- XBEGIN instruction does not have fencing semantics
  - *but, upon abort, all memory updates inside RTM region are invisible*
  - *same semantics as LOCK-prefixed instructions but without the cost*
- Intel provides no guarantee that the RTM region will eventually commit

TM: Benefits

- Simpler concurrent programming

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                       # commit txn
    }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Benefits

- Simpler concurrent programming
- Fewer concurrency bugs

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                      # commit txn
    }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Benefits

- Simpler concurrent programming
- Fewer concurrency bugs
- Improved scalability

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                     # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Benefits

- Simpler concurrent programming
- Fewer concurrency bugs
- Improved scalability
- Smoother composition

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                        # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Benefits

- Simpler concurrent programming
- Fewer concurrency bugs
- Improved scalability
- Smoother composition
- Reduced latency in low-contention scenarios

```
using System.Transactions;

int sharedResource = 0;

try {
    using (TransactionScope scope = new TransactionScope()) { # start txn

        settings.Update("Brightness", "80");
        settings.Update("Volume", "60");
        settings.Update("NightMode", "Enabled");

        scope.Complete();                                       # commit txn
   }
} catch (Exception ex) { # something went wrong, txn aborted
    ...
}
```

TM: Limitations

- Inherent in the tension between high / low levels of abstraction
- Long-running txns are more likely to abort
- Poor interaction with non-transactional resources (e.g., I/O)
- Interacting with legacy or non-txnal code
- Hard to debug
- Unpredictable performance
- Limited HW support

Recap

- Transactions in real life => transaction abstraction
- True transactions = ACID
- Can nest transactions (but not trivially)
- Transactional memory updates

Programming with Intel TSX

OPTIONAL

Intel TSX: XABORT

- abort the execution of an RTM region explicitly
- takes an 8-bit immediate argument for status code (goes into `eax`)

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}


__inline void _xend() { __asm { xend } }
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}


__inline void _xend() { __asm { xend } }


__inline void _xabort() { __asm { xabort } }
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}

__inline void _xend() { __asm { xend } }

__inline void _xabort() { __asm { xabort } }
```

Q: Can we pass to xbegin the address of some fallback code other than the instruction immediately following xbegin?
A: In principle yes, but keep in mind that, upon reaching that code, the registers and memory are restored to their state just prior to executing xbegin. The easiest is to transfer control as in the example here; if control is transferred elsewhere, then you will have to explicitly handle the discrepancies between the actual and expected state at that point. If you don't write the machine code directly, then the compiler will have, e.g., allocated variables to registers in a way that getting to that fallback code with the register and memory state of xbegin will confuse the program and exhibit undefined behavior.

```c
unsigned status;

// Start a transactional region
if ((status = _xbegin()) == _XBEGIN_STARTED) {
    // Transactional code goes here
    // This block of code will run in a transactional context

    // You would typically access shared data here

    // End the transactional region
    _xend();
    printf("Transaction successfully committed.\n");
} else {
    // Transaction failed
    if (status & _XABORT_EXPLICIT) {
        printf("Transaction explicitly aborted with code %x.\n", _XABORT_CODE(status));
    } else if (status & _XABORT_RETRY) {
        printf("Transaction failed but is retryable.\n");
    } else {
        printf("Transaction failed with status code %x.\n", status);
    }
    // Handle the failed transaction
    // Fallback to a non-transactional alternative or retry logic can go here
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG(“couldn't update global variable”);
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move    eax, _XBEGIN_STARTED
    xbegin _txnL1
    _txnL1:
    move    status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
     move    eax, _XBEGIN_STARTED
     xbegin _txnL1
     _txnL1:
     move    status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move   eax, _XBEGIN_STARTED
    xbegin _txnL1
    _txnL1:
    move   status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move   eax, _XBEGIN_STARTED
    xbegin _txnL1
    _txnL1:
    move   status, eax
  }
  return status;
}
```

```
while (1) {
  unsigned status = _xbegin(); // start transaction
  if (status == _XBEGIN_STARTED) {
    (*g)++; // non atomic increment of shared global variable
    /... do more stuff .../
    _xend();
    break; // break on success
  } else if (status == _XABORT_RETRY) {
    // try again
  } else {
    // fallback path
    LOG("couldn't update global variable");
  }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
     move    eax, _XBEGIN_STARTED
     xbegin _txnL1
     _txnL1:
     move    status, eax
  }
  return status;
}
```

1

2

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move   eax, _XBEGIN_STARTED
    xbegin _txnL1
    _txnL1:
    move   status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

1

2

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

1

2

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
#pragma omp parallel for
for (int i=0; i < N; i++) {
   int mygroup = group[i];
   if (_xbegin() == _XBEGIN_STARTED) {
      sums[mygroup] += data[i];
      _xend();
   } else {
      #pragma omp critical
      {
         sums[mygroup] += data[i];
      }
   }
}
```

Courtesy of https://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions

Loop iterations can be
spread over available threads

```
#pragma omp parallel for
for (int i=0; i < N; i++) {
   int mygroup = group[i];
   if (_xbegin() == _XBEGIN_STARTED) {
      sums[mygroup] += data[i];
      _xend();
   } else {
      #pragma omp critical
      {
         sums[mygroup] += data[i];
      }
   }
}
```

Loop iterations can be
spread over available threads

```
#pragma omp parallel for
for (int i=0; i < N; i++) {
   int mygroup = group[i];
   if (_xbegin() == _XBEGIN_STARTED) {
      sums[mygroup] += data[i];
      _xend();
   } else {
      #pragma omp critical
      {
         sums[mygroup] += data[i];
      }
   }
}
```

Loop iterations can be
spread over available threads

```
for (int i=0; i < N; i++) {
   int mygroup = group[i];
   if (_xbegin() == _XBEGIN_STARTED) {
      if (!fallback_mutex.is_acquired()) {  // add the mutex to the txn read set
         sums[mygroup] += data[i];
      } else {
         _xabort(MY_ABORT_ERROR_CODE);
      }
      _xend();
   } else {
      fallback_mutex.acquire();
      sums[mygroup] += data[i];
      fallback_mutex.release();
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

Q: Can we pass to xbegin the address of some fallback code other than the instruction immediately following xbegin?
A: In principle yes, but keep in mind that, upon reaching that code, the registers and memory are restored to their state just prior to executing xbegin. The easiest is to transfer control as in the example here; if control is transferred elsewhere, then you will have to explicitly handle the discrepancies between the actual and expected state at that point. If you don't write the machine code directly, then the compiler will have, e.g., allocated variables to registers in a way that getting to that fallback code with the register and memory state of xbegin will confuse the program and exhibit undefined behavior.

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}

__inline void _xend() { __asm { xend } }
```

Q: Can we pass to xbegin the address of some fallback code other than the instruction immediately following xbegin?
A: In principle yes, but keep in mind that, upon reaching that code, the registers and memory are restored to their state just prior to executing xbegin. The easiest is to transfer control as in the example here; if control is transferred elsewhere, then you will have to explicitly handle the discrepancies between the actual and expected state at that point. If you don't write the machine code directly, then the compiler will have, e.g., allocated variables to registers in a way that getting to that fallback code with the register and memory state of xbegin will confuse the program and exhibit undefined behavior.

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, 0xFFFFFFFF   // put _XBEGIN_STARTED in eax
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}


__inline void _xend() { __asm { xend } }


__inline void _xabort() { __asm { xabort } }
```

Q: Can we pass to xbegin the address of some fallback code other than the instruction immediately following xbegin?
A: In principle yes, but keep in mind that, upon reaching that code, the registers and memory are restored to their state just prior to executing xbegin. The easiest is to transfer control as in the example here; if control is transferred elsewhere, then you will have to explicitly handle the discrepancies between the actual and expected state at that point. If you don't write the machine code directly, then the compiler will have, e.g., allocated variables to registers in a way that getting to that fallback code with the register and memory state of xbegin will confuse the program and exhibit undefined behavior.

```c
unsigned status;

// Start a transactional region
if ((status = _xbegin()) == _XBEGIN_STARTED) {
    // Transactional code goes here
    // This block of code will run in a transactional context

    // You would typically access shared data here

    // End the transactional region
    _xend();
    printf("Transaction successfully committed.\n");
} else {
    // Transaction failed
    if (status & _XABORT_EXPLICIT) {
        printf("Transaction explicitly aborted with code %x.\n", _XABORT_CODE(status));
    } else if (status & _XABORT_RETRY) {
        printf("Transaction failed but is retryable.\n");
    } else {
        printf("Transaction failed with status code %x.\n", status);
    }
    // Handle the failed transaction
    // Fallback to a non-transactional alternative or retry logic can go here
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG(“couldn't update global variable”);
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move    eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move    status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
   unsigned status;
   __asm {
      move   eax, _XBEGIN_STARTED
      xbegin _txnL1
      _txnL1:
      move   status, eax
   }
   return status;
}
```

1

2

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
    move   eax, _XBEGIN_STARTED
    xbegin _txnL1
    _txnL1:
    move   status, eax
  }
  return status;
}
```

1

2

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG(“couldn't update global variable”);
   }
}
```

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
     move    eax, _XBEGIN_STARTED
     xbegin _txnL1
     _txnL1:
     move    status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG("couldn't update global variable");
   }
}
```

1

2

```
__inline unsigned int _xbegin() {
  unsigned status;
  __asm {
     move    eax, _XBEGIN_STARTED
     xbegin _txnL1
     _txnL1:
     move    status, eax
  }
  return status;
}
```

```
while (1) {
   unsigned status = _xbegin(); // start transaction
   if (status == _XBEGIN_STARTED) {
      (*g)++; // non atomic increment of shared global variable
      /... do more stuff .../
      _xend();
      break; // break on success
   } else if (status == _XABORT_RETRY) {
     // try again
   } else {
     // fallback path
     LOG(“couldn't update global variable”);
   }
}
```