Example to understand Object Pooling

1. Object per Client (Own Vehicle)

2. Object per request (Cab Vehicle)

3. Shareable cum Stateless Objects (Public Transport)

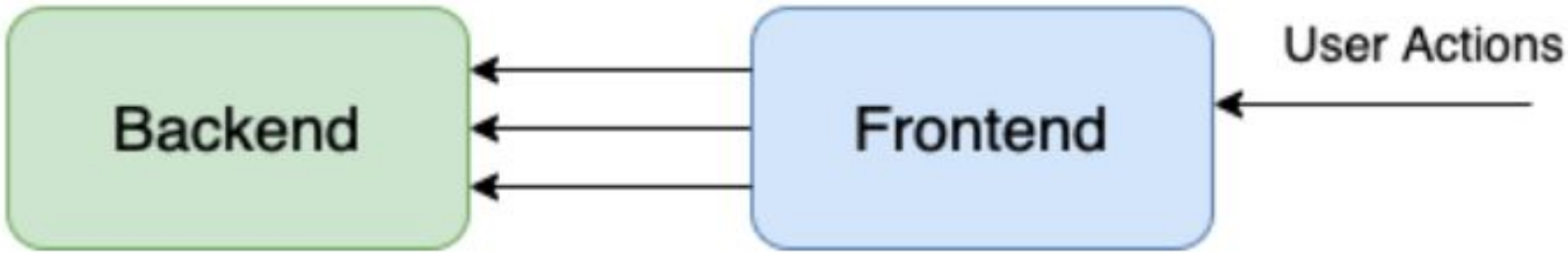
Stateless means Objects with out state and session information

Any Object to any request of any client

i.e. Objects are independent of Client and Request

Object Pool: Basic Idea

- An object pool is a set of initialized objects that are recycled rather than allocated and destroyed on demand.
- A client requests an object from the pool and performs operations on the returned object. When the client has finished with an object, it returns it to the pool rather than destroying it.
- Consecutive requests need not routed to the same instance of the object. i.e. Any request route to any instance(Stateless)
- Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low.
- The pooled object is obtained in predictable time, which makes this pattern useful for real-time systems.
- Size of the pool decide the performances



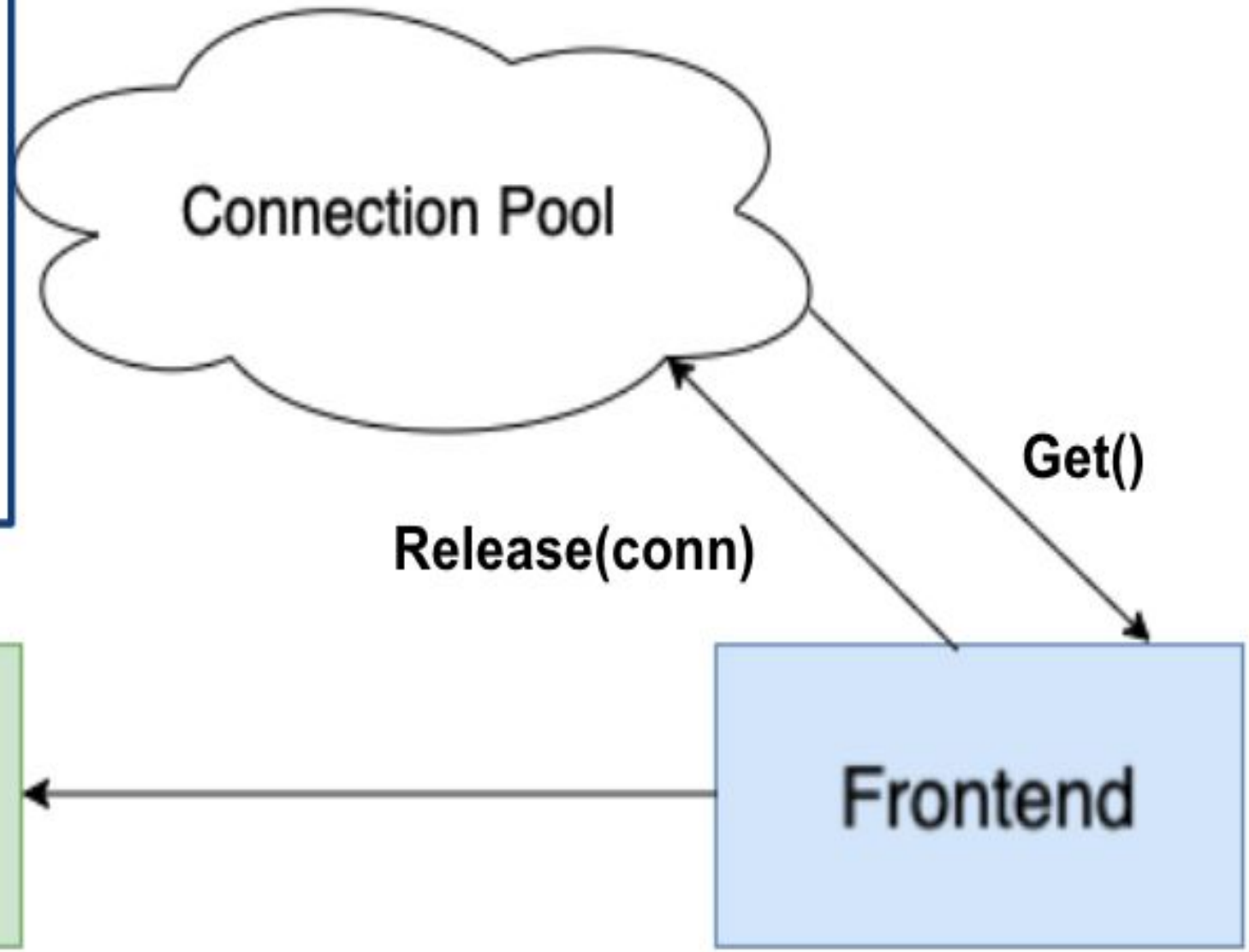
Inefficient due to cost overhead of

1. Teardown
2. Opening connection

For every request

For each requests, just Get existing conn from pool. No more

1. Teardown
2. Opening connection



Example

1. `myShoes = Shelf.acquireShoes();`

SHELF (OBJECT POOL)

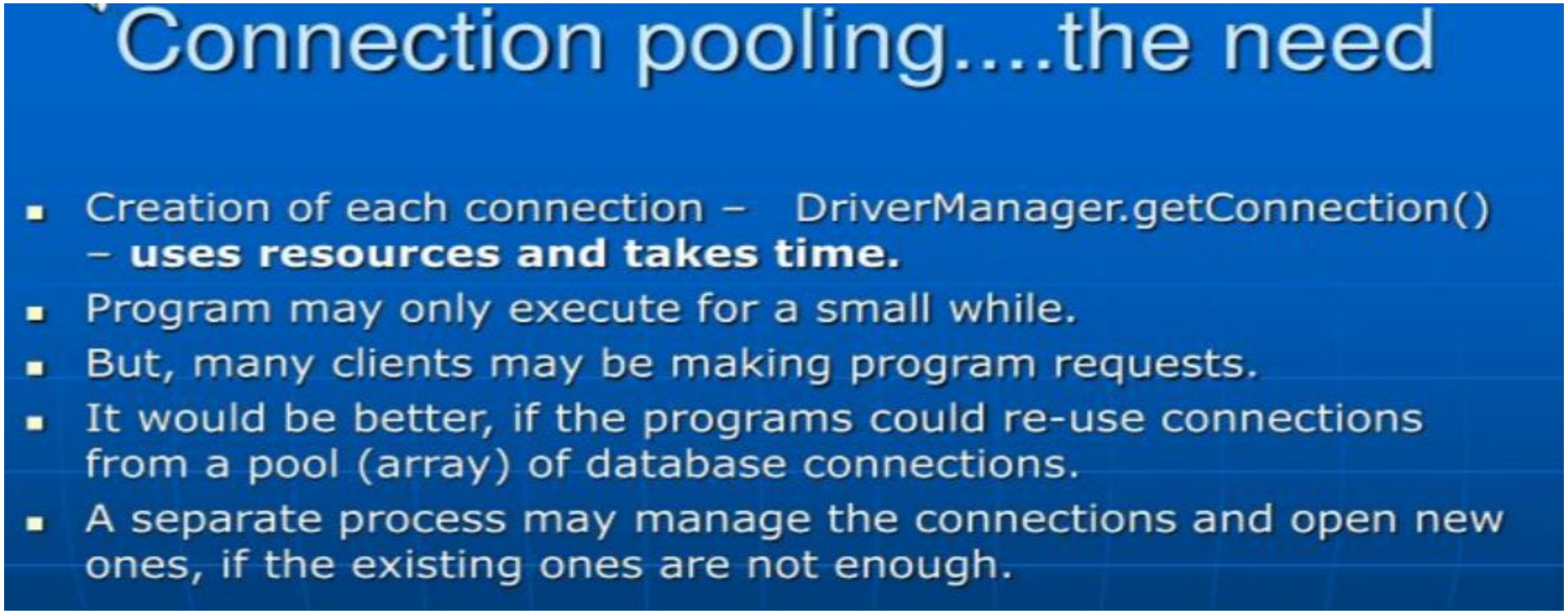
2. `client.wear(myShoes);`

3. `client.play();`

4. `Shelf.releaseShoes(myShoes);`

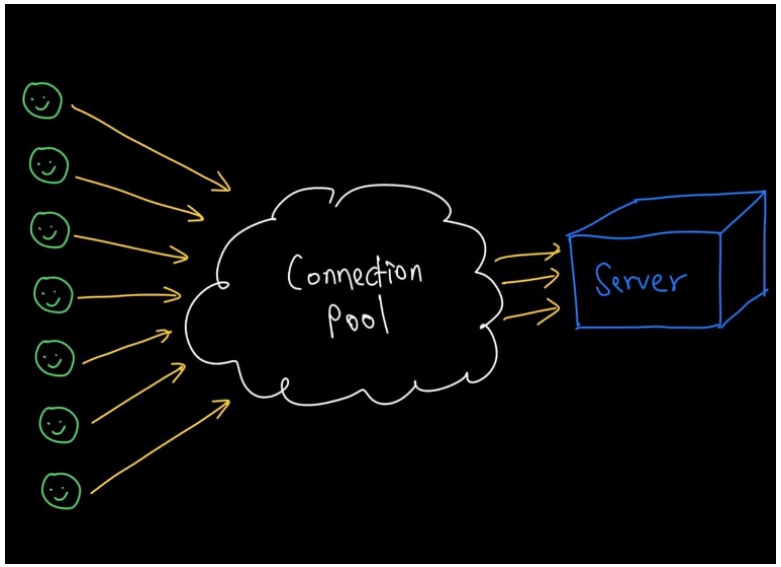
What Is a Connection Pool?

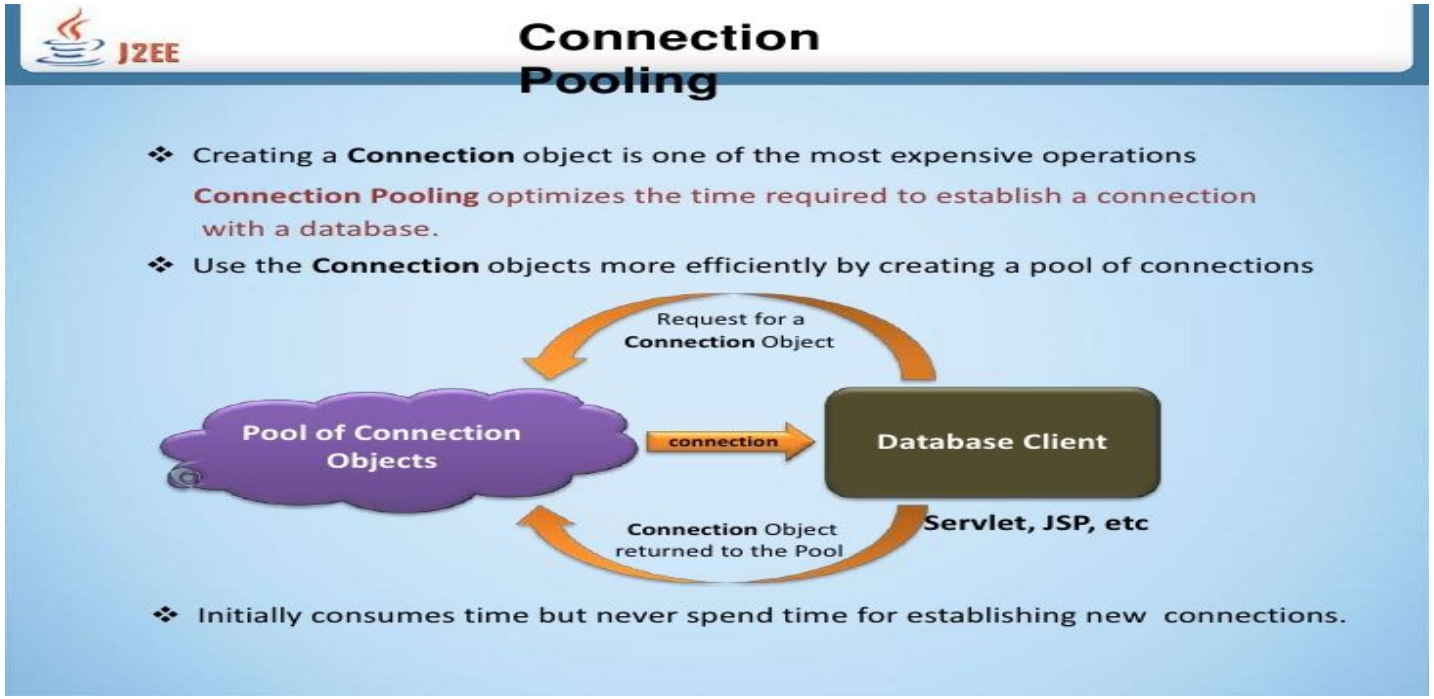
- A pool (or pooling) is a software component that maintains a set of pre-configured resources.
- Clients of a pool (Java methods, classes, or in general, other software components) can “borrow” a resource, use it, and return it to the pool so that other clients can use it.
- In this case, the resources are database connections, hence the name database connection pool.
- This kind of pool keeps database connections ready to use, that is, JDBC Connection objects.
- Typically, Java threads request a Connection object from the pool, perform CRUD operations, and close the connection, effectively returning it to the pool.
- In fact, threads are another common resource that uses pools



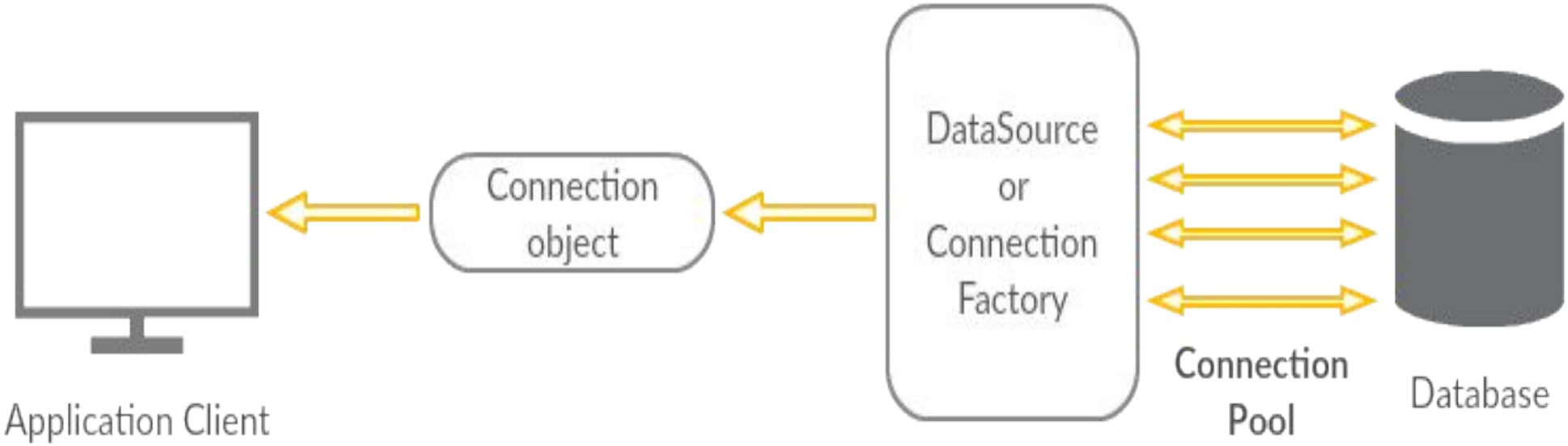
Why do we need Connection Pooling?

- An object pool is a set of limited resources, such as connections, that can be reserved for use by clients and then returned to the pool (for probable reuse) when the object is no longer needed.
- Reserving and returning pooled objects avoids the overhead of separately creating and destroying an object each time a client requests it.
- Multiple object pools can be used.
- For example, one object pool might contain database connection objects

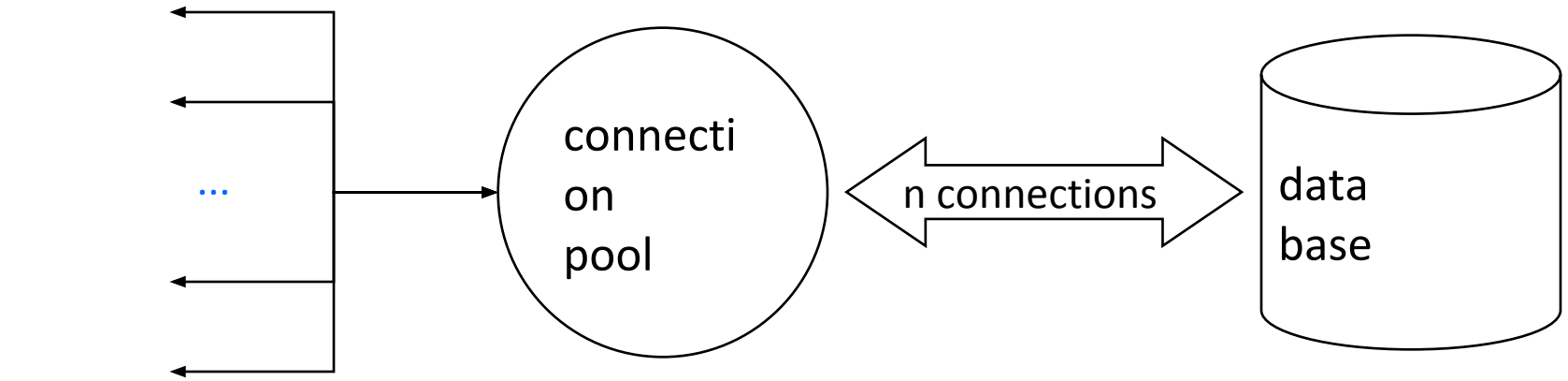




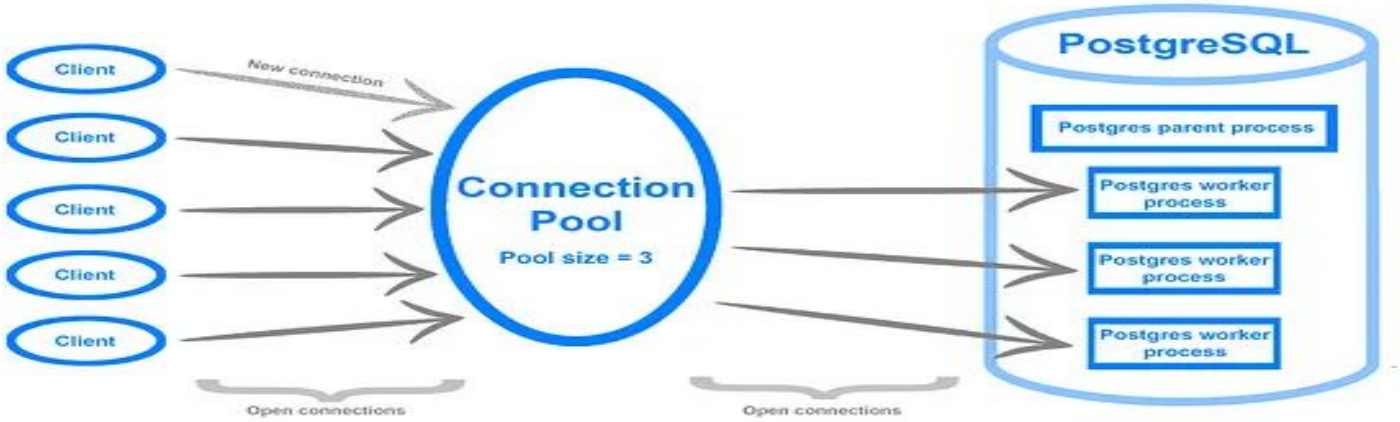
How Connection Pool Works?



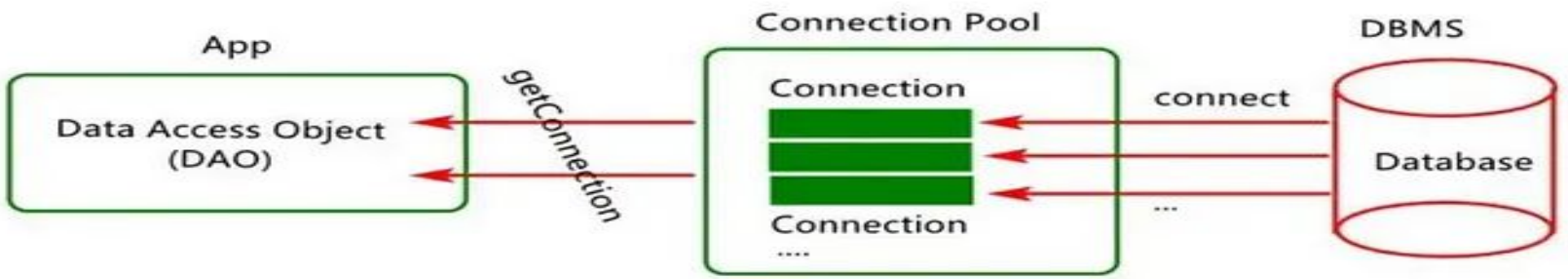
Role of a Connection Pool



max n simultaneous
connections



With Connection Pool



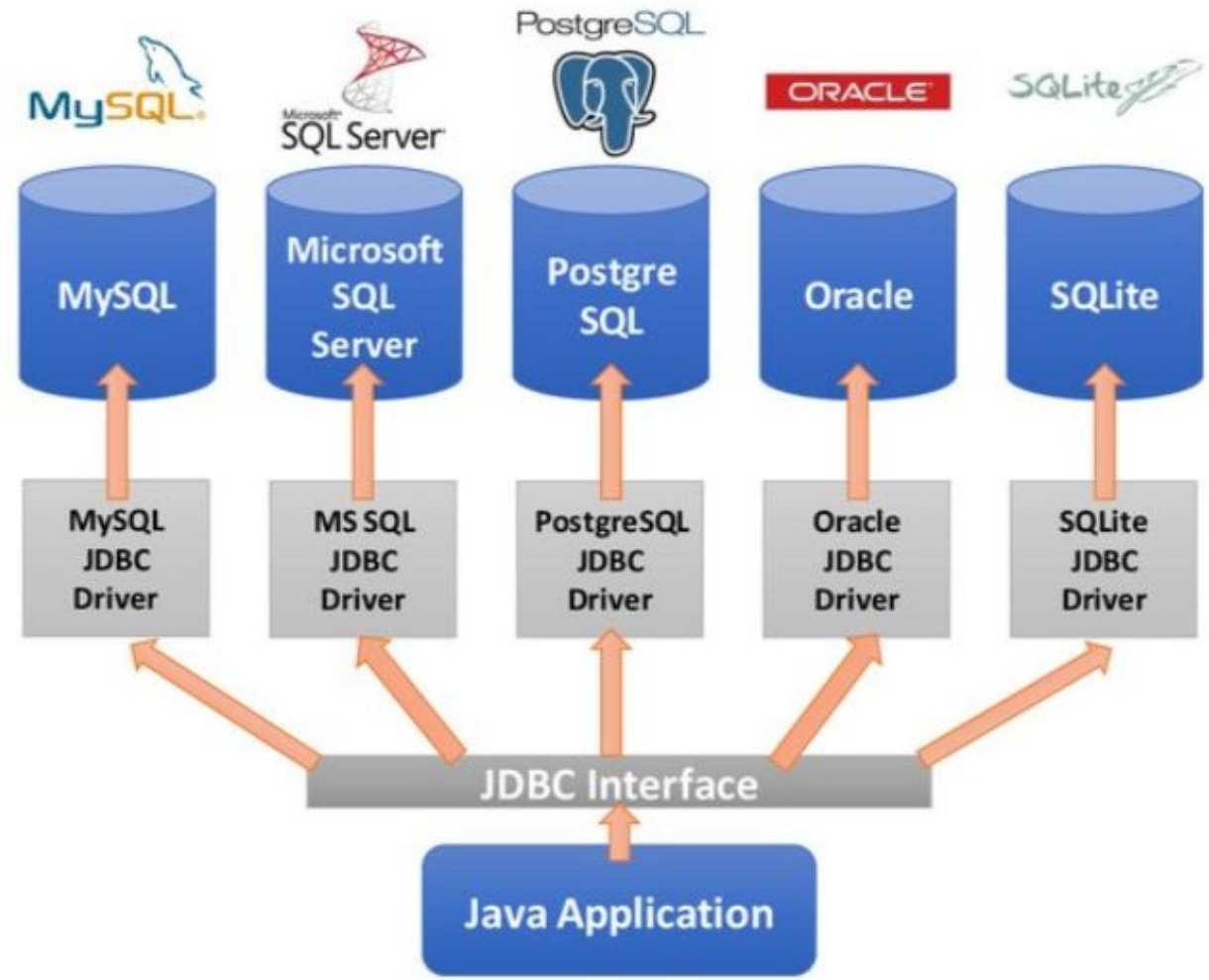
A connection pool is used to solve the exact problem, it creates a pool of reusable connections ahead of time to improve the application performance.

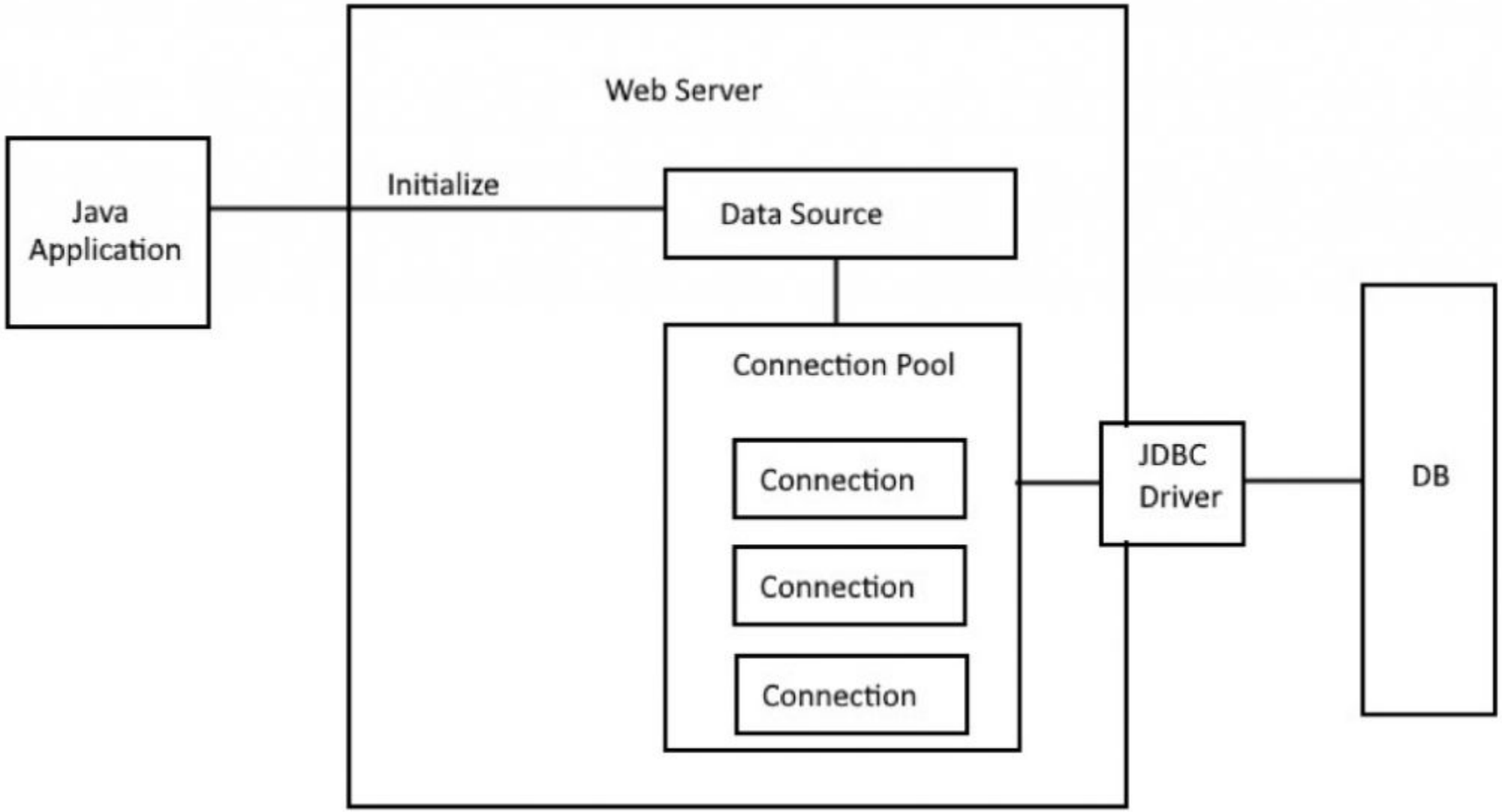
Whenever the application starts a pool of Connections is created with the database.

These connections are managed by a Pool Connection Manager.

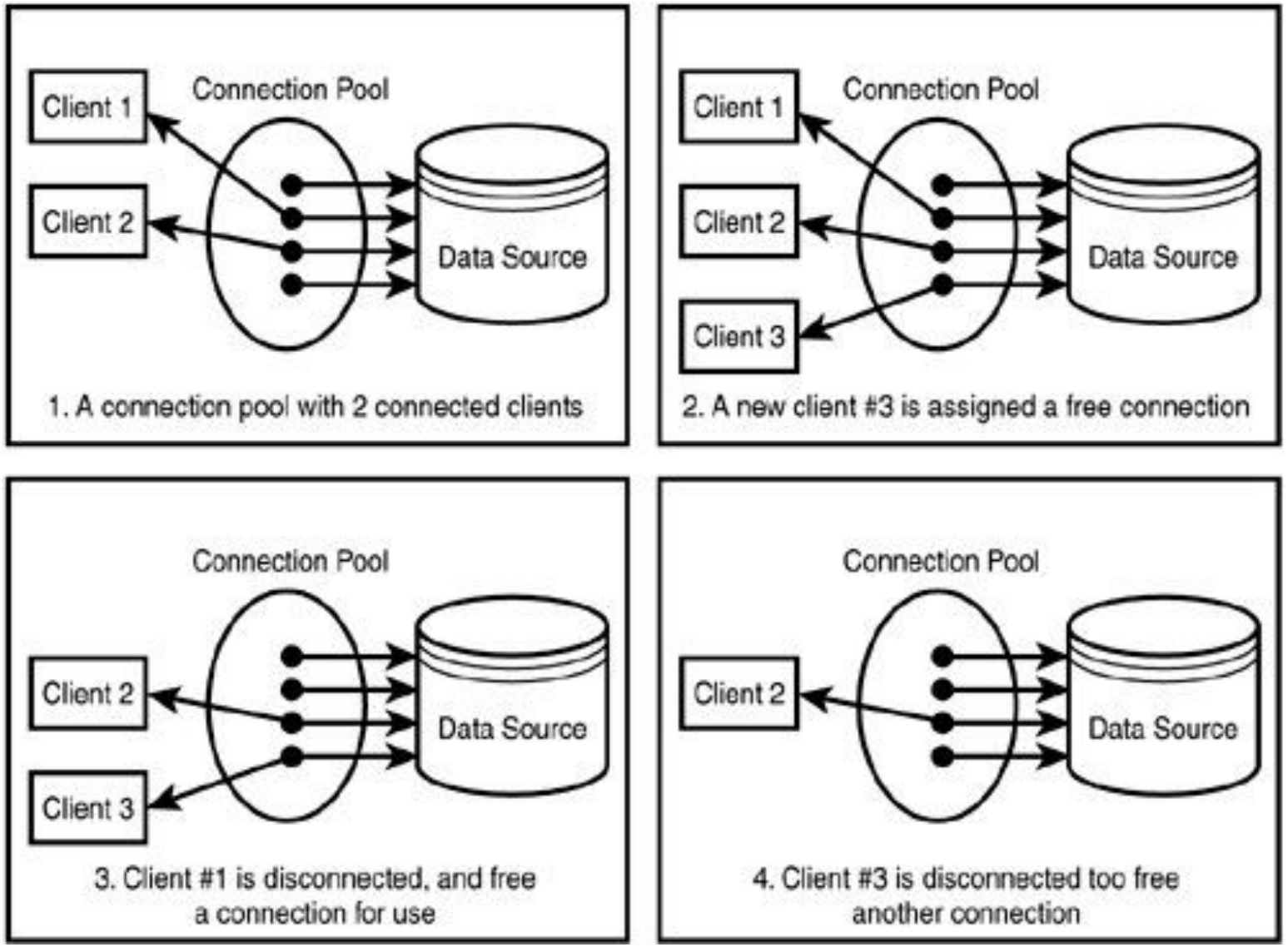
The latter is responsible for managing the lifecycle of a connection.

Driver


MySQL
Microsoft SQL Server
PostgreSQL
ORACLE
SQLite
MySQL
Microsoft SQL Server
Postgre SQL
Oracle
SQLite
MySQL JDBC Driver
MS SQL JDBC Driver
PostgreSQL JDBC Driver
Oracle JDBC Driver
SQLite JDBC Driver
JDBC Interface
Java Application

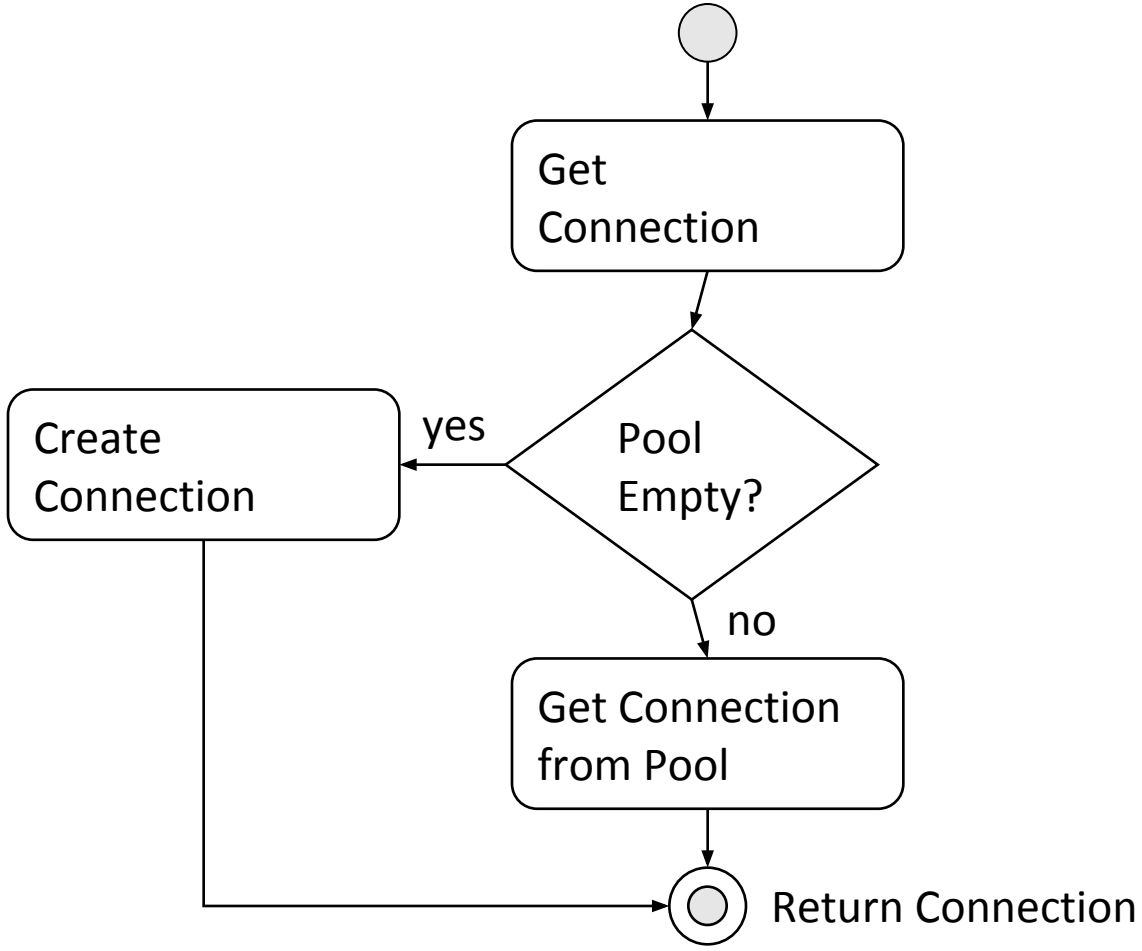

Web Server
Java Application
Initialize
Data Source
Connection Pool
Connection
Connection
Connection
JDBC Driver
DB

Connect pool usage scenario

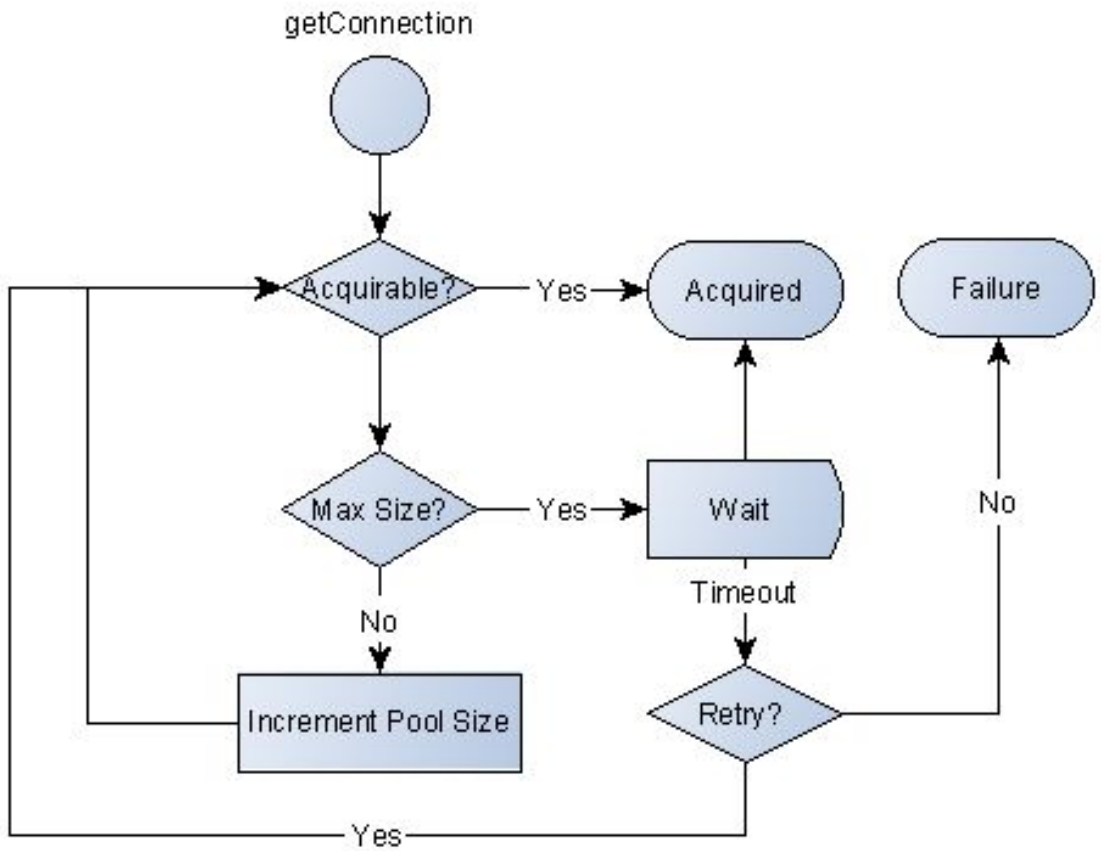


Database Connection Pooling

Cache connections so that they can be reused.



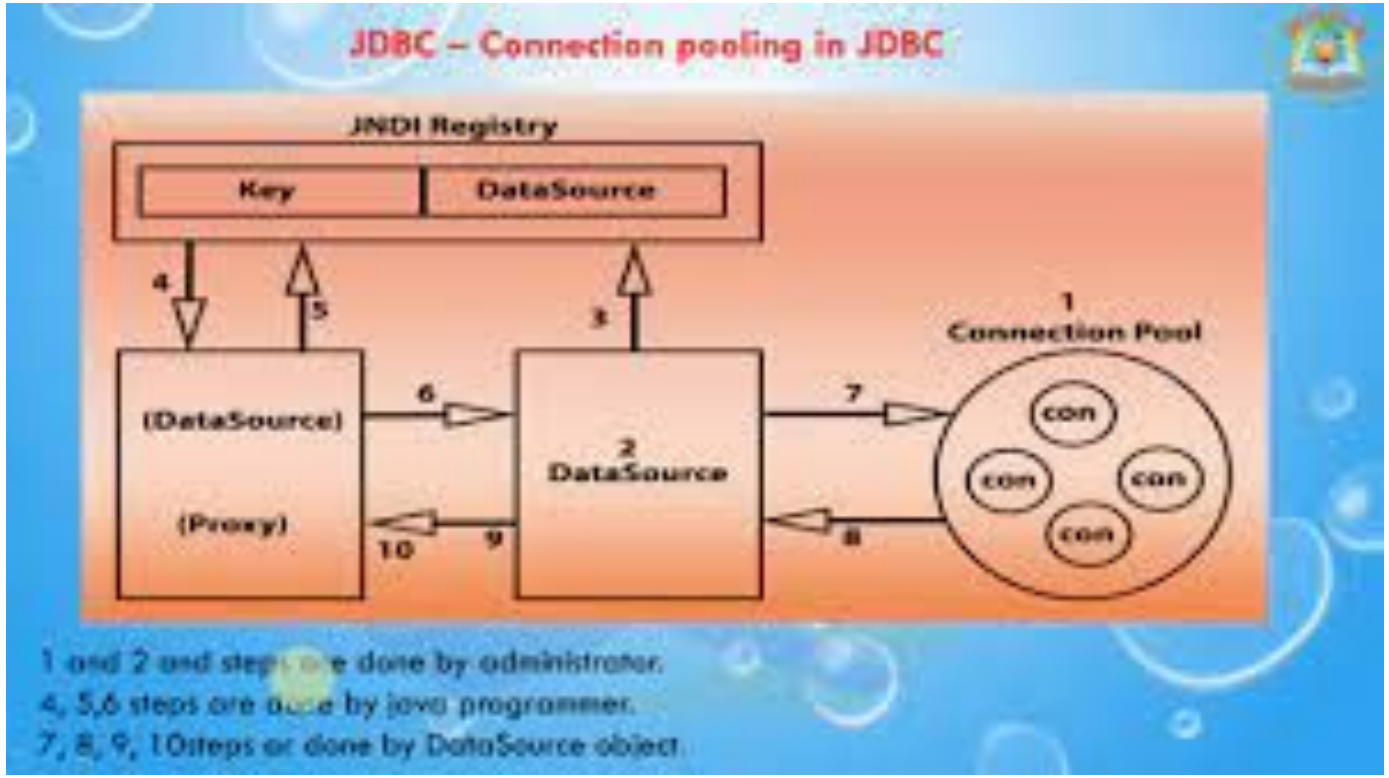
Connection pooling working principles

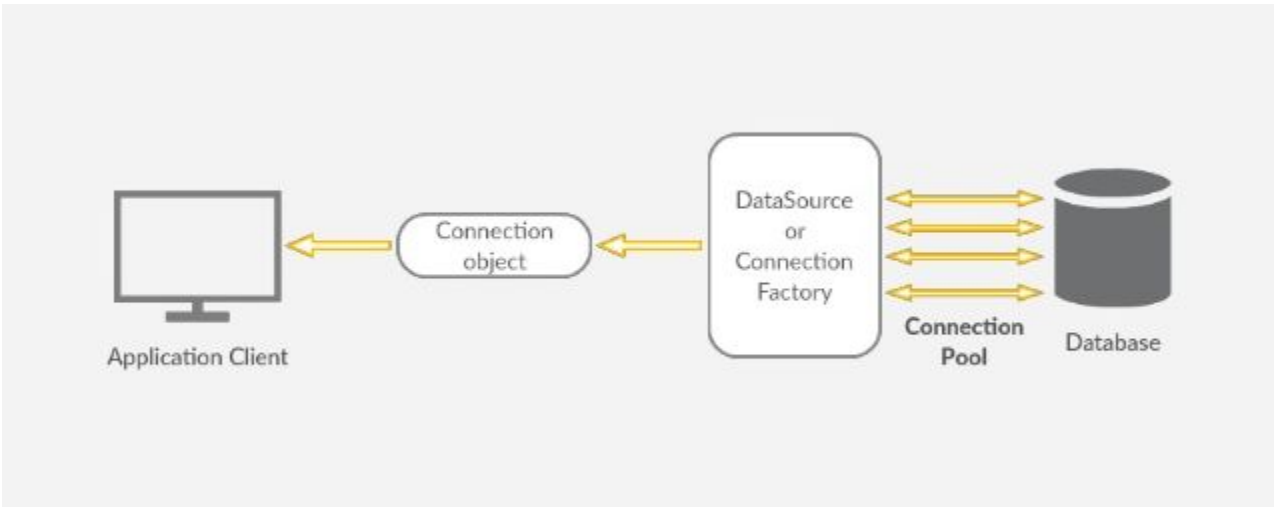


JDBC IMPLEMENTATION MECHANISM

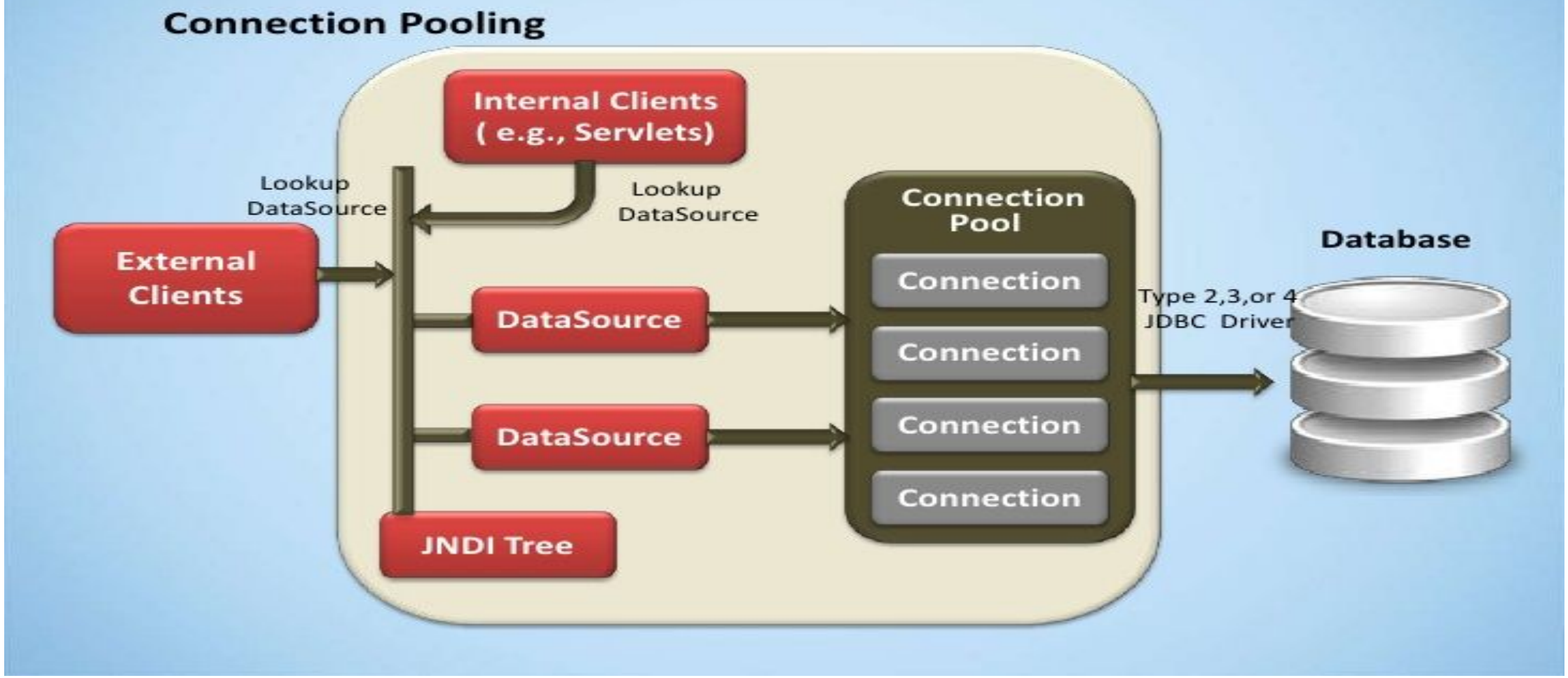
1. DriverManager
   - hampers the application performance as the connections are created/closed in java classes.
   - does not support connection pooling.

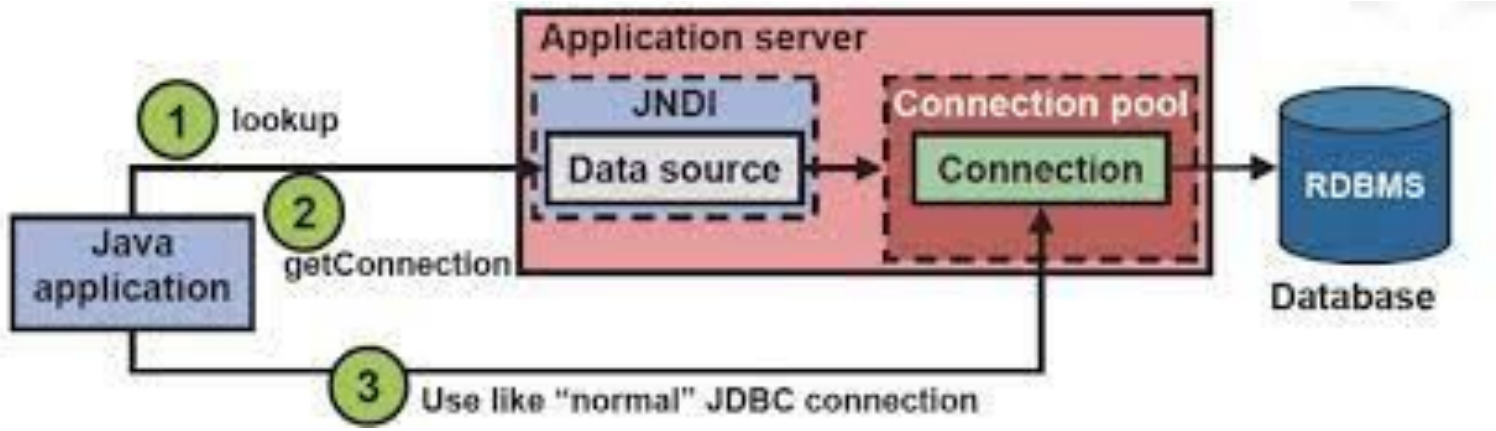
2. DataSource
   - improves application performance as connections are not created/closed within a class, they are managed by the application server and can be fetched while at runtime.
   - it provides a facility creating a pool of connections
   - helpful for enterprise applications

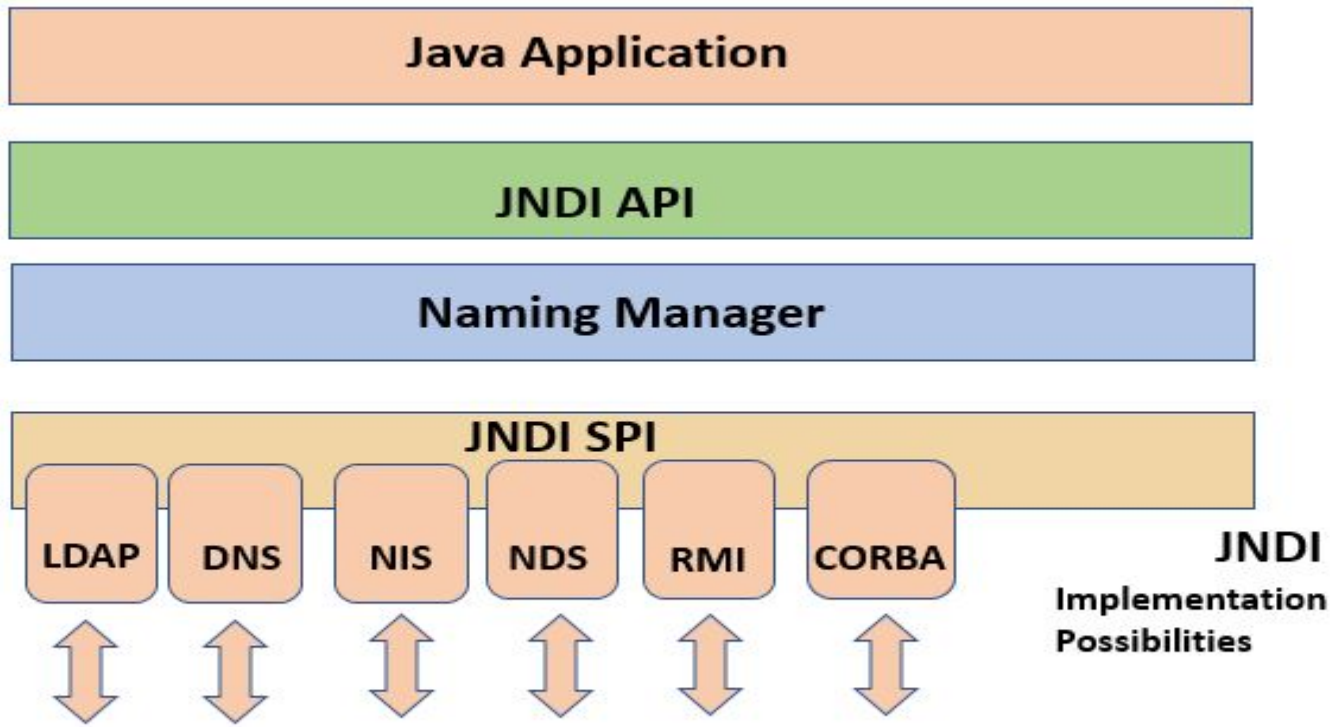

JDBC – Connection pooling in JDBC
JNDI Registry
Key
DataSource
4
5
3
1
Connection Pool
(DataSource)
(Proxy)
6
2
DataSource
7
10
9
8
con
con
con
con
1 and 2 and step are done by administrator.
4, 5,6 steps are done by java programmer.
7, 8, 9, 10steps are done by DataSource object


Application Client
Connection object
DataSource or Connection Factory
Connection Pool
Database

Connection Pooling







A connection pool provides the following benefits for an Enterprise Applications

1. Reduces the number of times new connection objects are created.
2. Promotes connection object reuse.
3. Quickens the process of getting a connection.
4. Reduces the amount of effort required to manually manage connection objects.
5. Minimizes the number of stale connections.
6. Controls the amount of resources spent on maintaining connections.

Dirty Read

- Dirty read is a read of uncommitted data.
- If a particular row is modified by another running application and not yet committed, we also run an application to read the same row with the same uncommitted data. This is the state we say it as a dirty read.

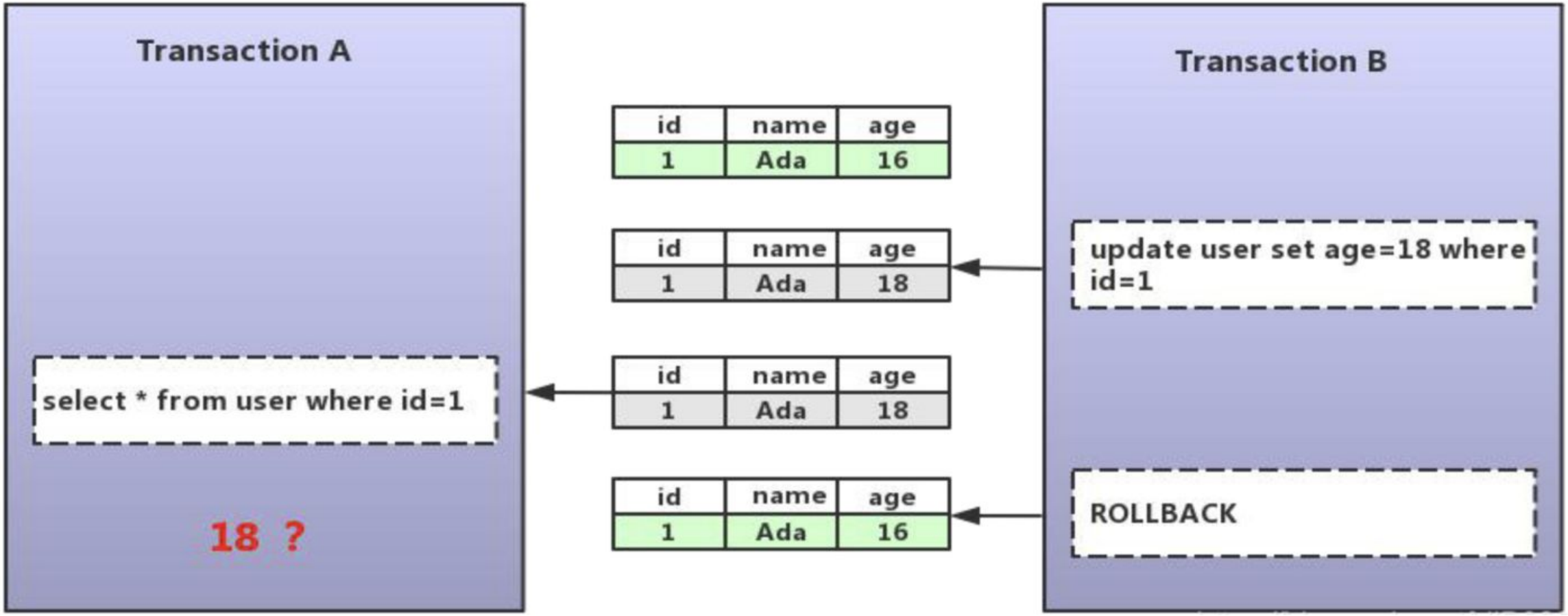
Example of dirty read problem

Example 1

- **Step 1** − Consider we have an online shopping system where users can buy and view the buyer products at the same time.
- **Step 2** − Let us suppose a case in which a user tries to buy a product , and as soon as the user buys the product then the count value in update stock will change immediately.
- **Step 3** − Let us consider that there were 10 items in stock, but now they are 9.
- **Step 4** − Moreover due to this transaction, there will also be communication with the billing gateway.
- **Step 5** − Meanwhile, if there is any other user who has also done a transaction at the same time, the new user will be able to see 9 items in the stock.
- **Step 6** − But, let us suppose that the first user was unable to complete his/her transactions due to some error or insufficient funds.
- **Step 7** − Then, in this case the transaction done by the first user will roll back and now the value in stock will be 10 again.
- **Step 8** − But, when the 2nd user was doing a transaction the no items in stocks were 9.
- **Step 9** − This is called DIRTY DATA and this whole problem is called the Dirty Problem.

Dirty read

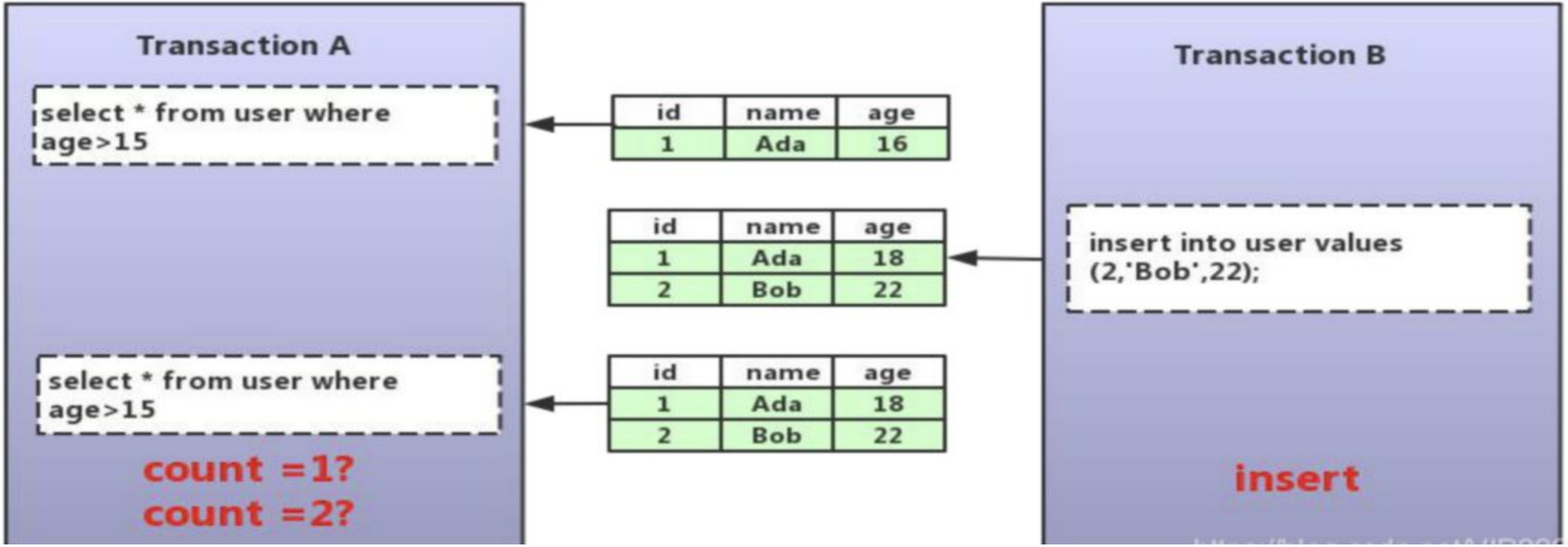
Uncommitted data is read.



Transaction B is rolled back at this time, then the second transaction A reads dirty data which age is 18

Phantom read

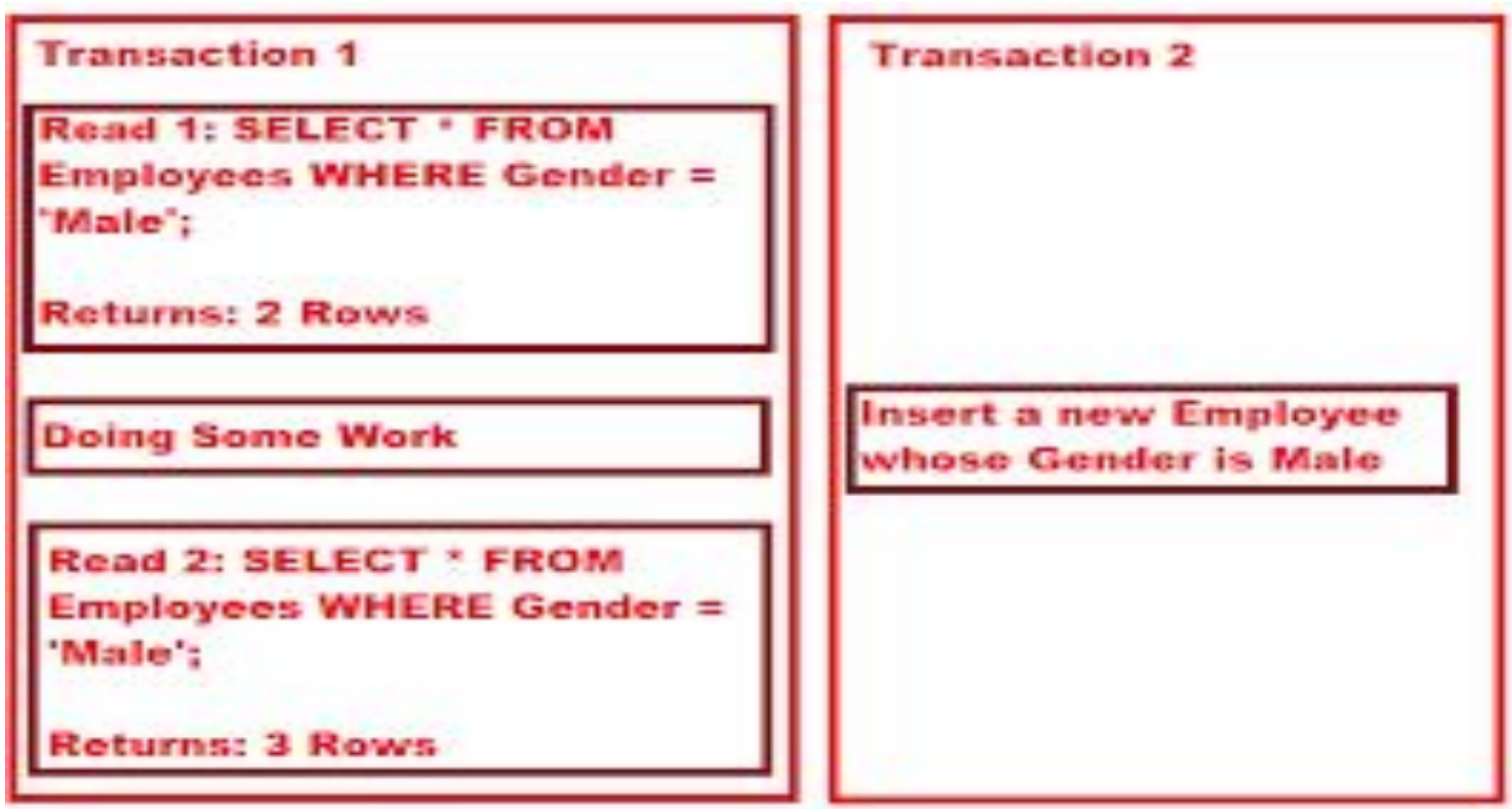
When the user reads records, another transaction inserts or deletes rows to the records being read. When the user reads the same rows again, a new “phantom” row will be found.



Transaction B inserts a new row where transaction A reads, then transaction A finds the total count of the result changes to 2

Non-Repeatable Read

Non repeatable read problem happens when one transaction reads the same data twice and another transaction updates that data in between the first and second read of transaction one.



Transaction isolation levels

Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

What is Transaction?

- In General, Transaction is agreement, Contract, exchange, understanding, or transfer of assets or cash that occurs between two or more parties and establish Legal Obligation.
- In commerce Perception, it is exchange of Goods and Services between buyers and Sellers.

- In computing, it is an event or process initiated or invoked by a user or Computer program, regarded as a single Unit of work and requiring a record to be generated for processing.
- In a secure transaction, such events are regarded as a single unit of work and must either be processed in their totality or rejected as a failed Transaction.

When do you need transactions

➡ When one business operation depends on another

- Banking - Withdrawing money from an account
- e-commerce - Completing a product purchase and a delivery request
- Inventory control - record a fulfillment of an order
- Manufacturing - Production line activities
- Government - Issuing a social security number

WHY DO WE NEED TRANSACTIONS?

Data must be in known state anytime

New order created → Payment received → Product shipped

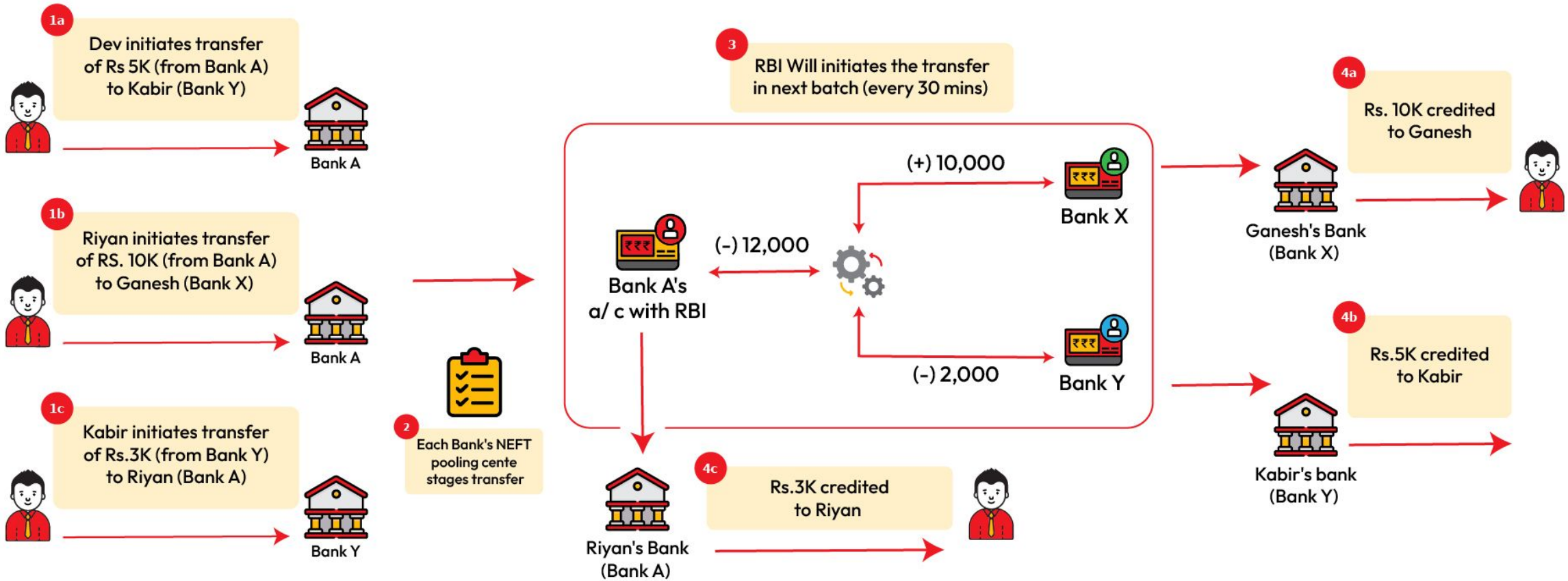
Application

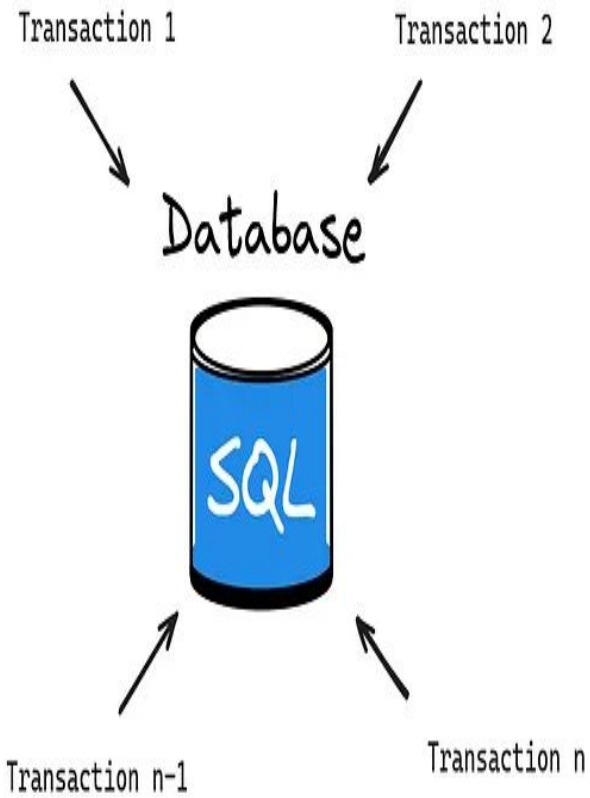
Business workflows

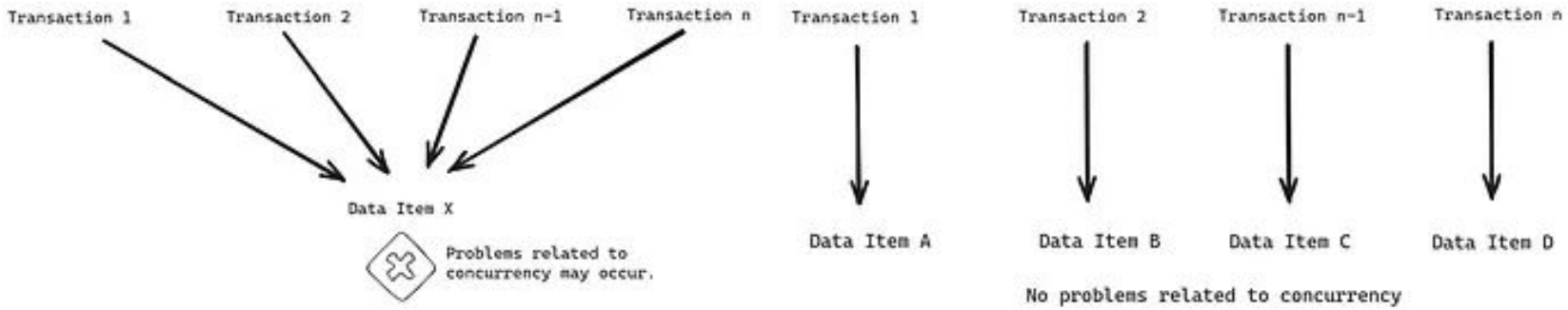
Infrastructure

Transactions in file systems, databases, message brokers, caches, application servers

NEFT (National Electronic Fund Transfer)







Transaction @ Database Context

HOW Transaction WORKS ?

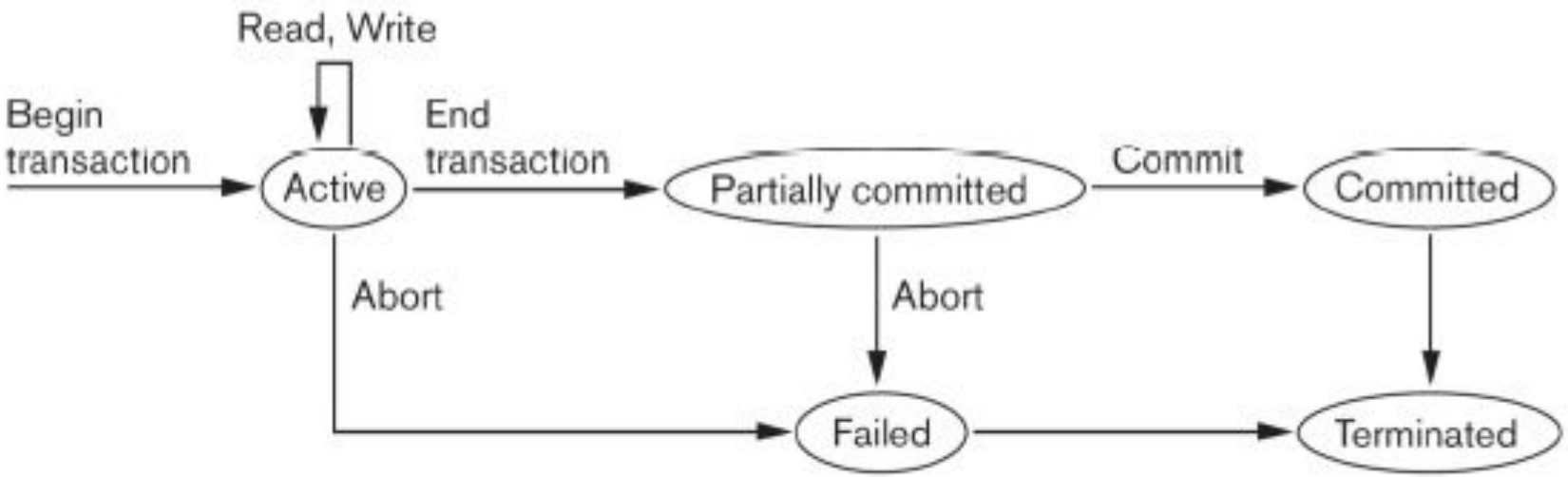
Transaction processing is based on **logging of States and Transitions**

2 Transaction and System Concepts (1)

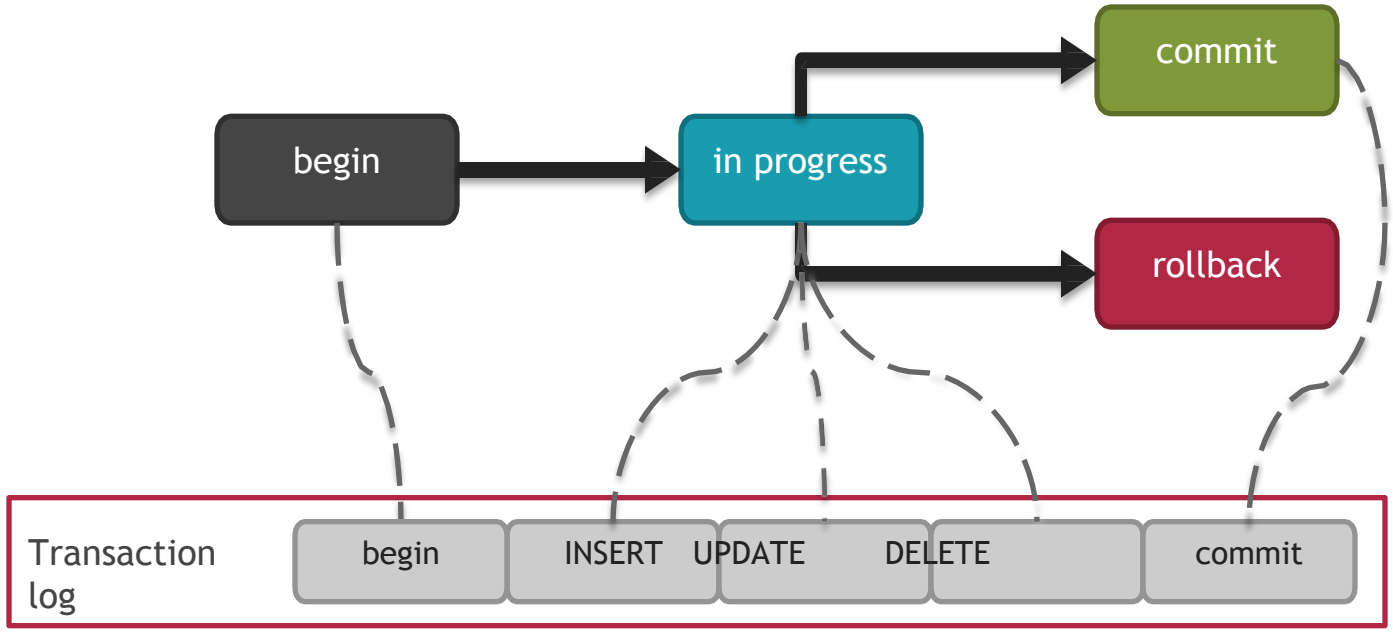
- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State Transition Diagram Illustrating the States for Transaction Execution

Figure 21.4
State transition diagram illustrating the states for transaction execution.



DATABASE TRANSACTION AS STATE MACHINE



- Always in known state
- States and transitions are logged
- Log can be used for recovery

What is Transaction ?

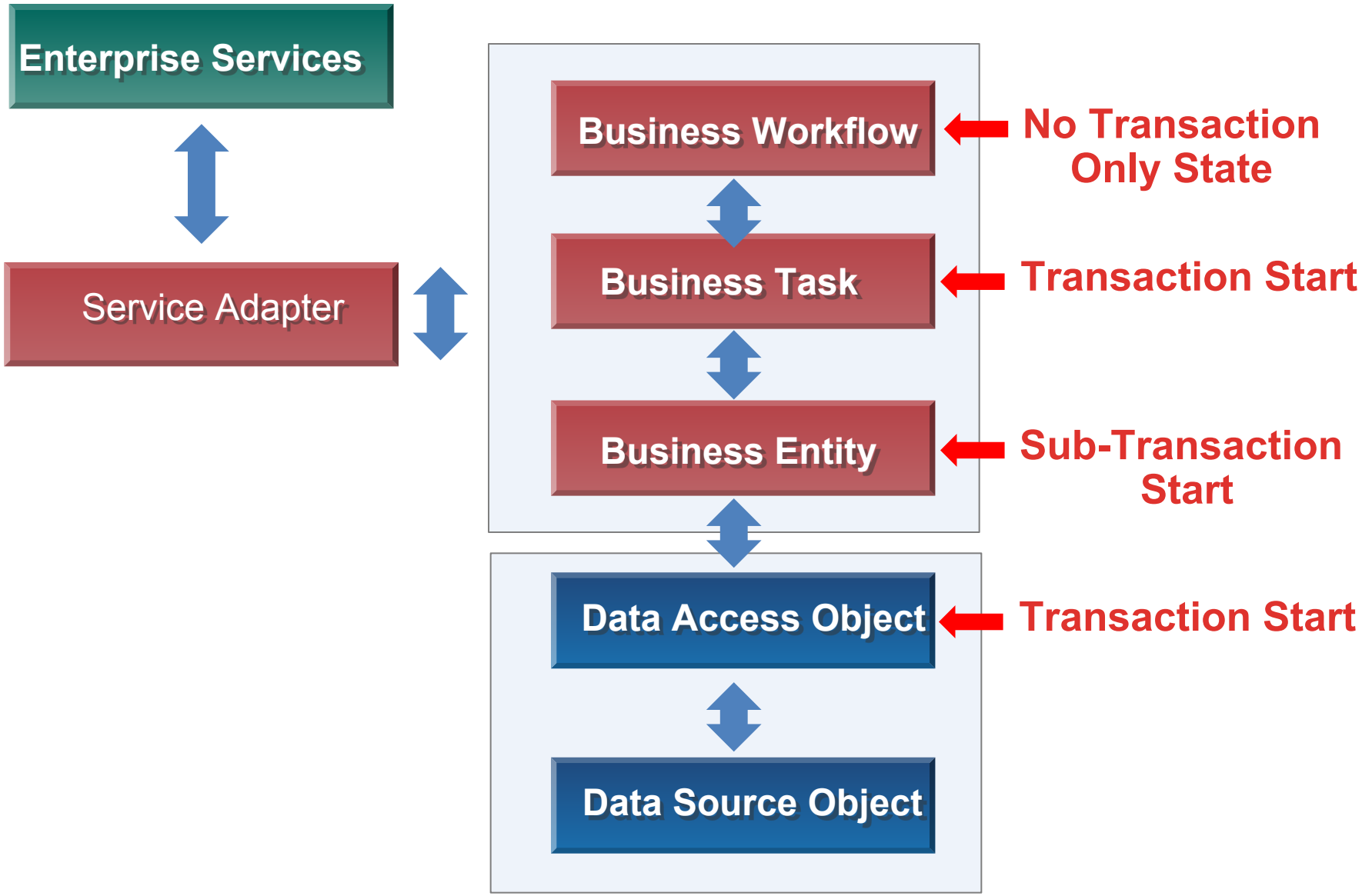
A transaction is a group of integral operations. These operations are either executed successfully as a whole or canceled as a whole.

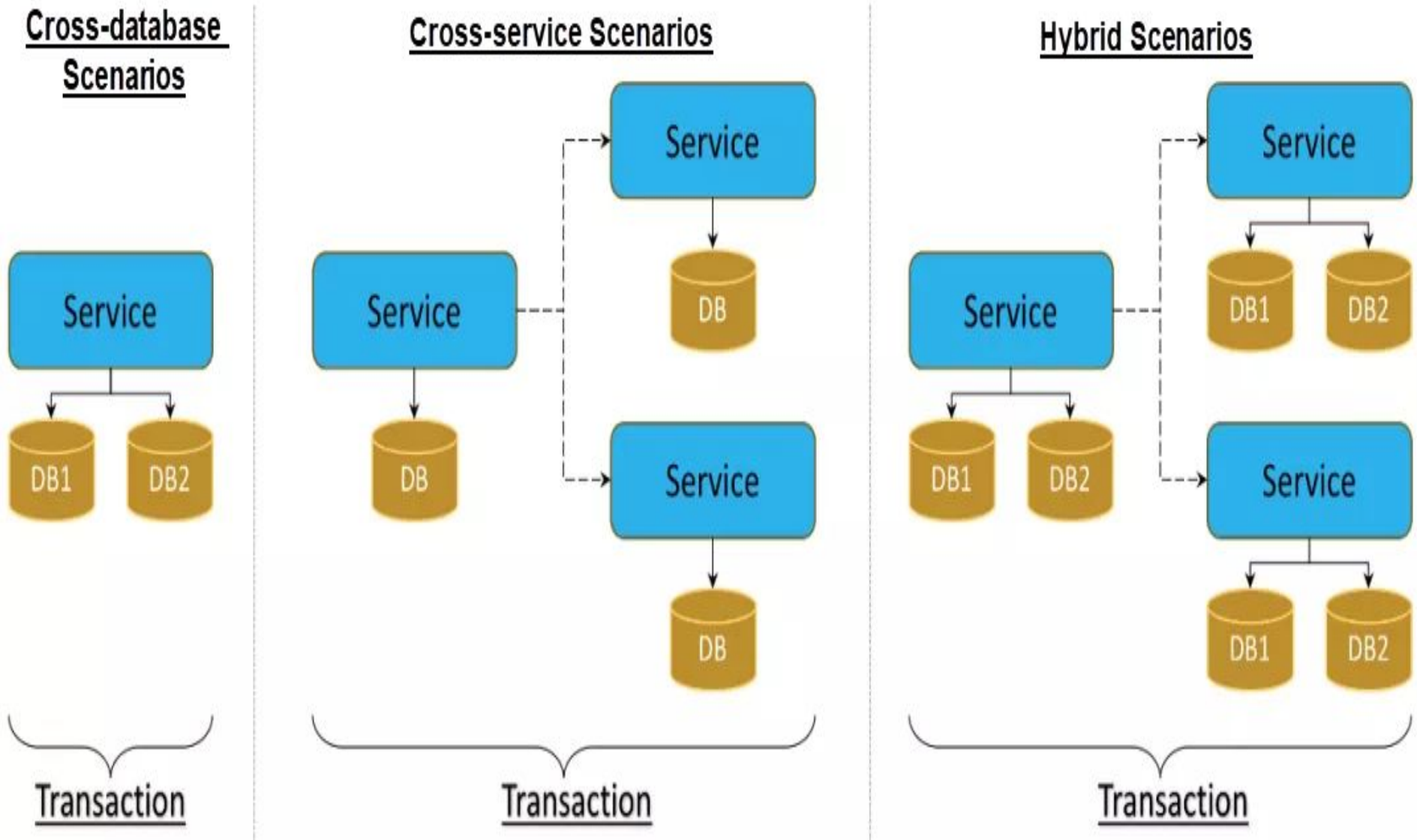
e.g. The most typical transaction scenario is a transfer between bank accounts. If account A wants to transfer RMB 100 to account B, RMB 100 will be deduced from A and added to B. In this scenario, a successful transfer can be considered only when amount changes are made to both accounts successfully.

More strictly, a transaction is defined to have the ACID features, which refer to atomicity, consistency, isolation, and durability.

1. **Atomicity**: All operations in a transaction must be successfully executed as a whole, or all operations must be canceled.
2. **Consistency**: For example, account A and account B have RMB 100 each. No matter how many times the two accounts transfer money to each other, the total balance of the two accounts are always RMB 200.
3. **Isolation**: indicates the extent of impact between concurrent transactions. It is also classified into uncommitted reads, committed reads, and repeatable reads.
4. **Durability**: After a transaction is completed, data changes made to the database are retained.

Levels of Transactions

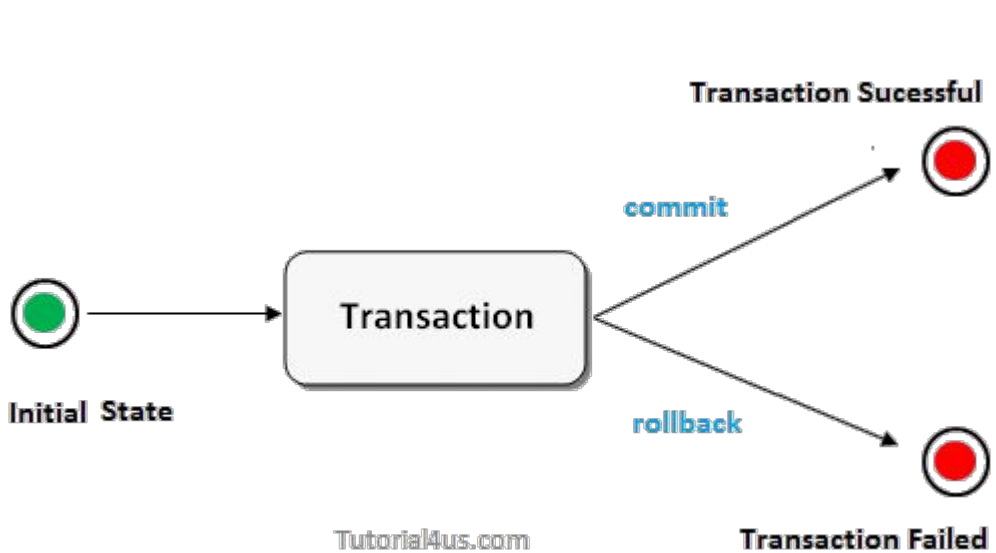


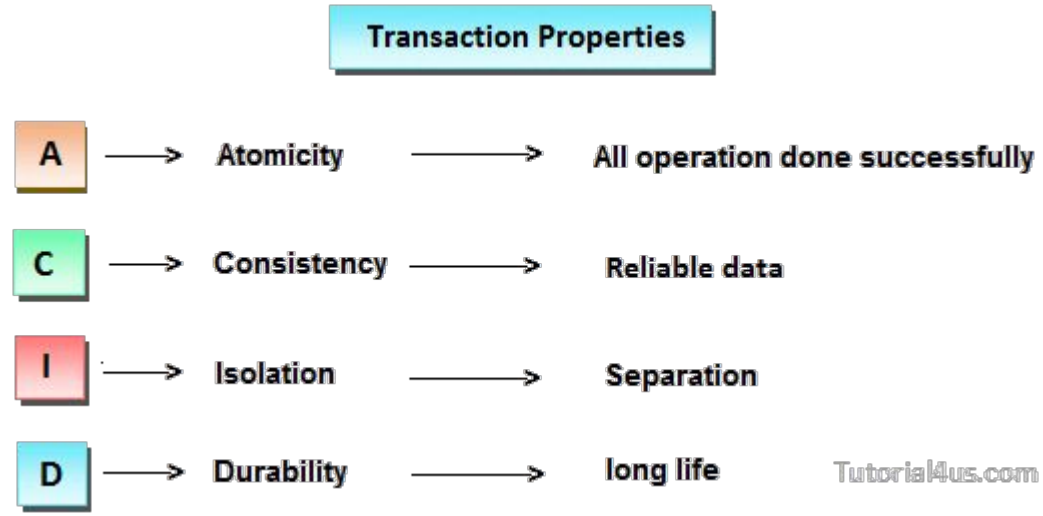

Cross-database Scenarios
Service
DB1
DB2
Transaction
Cross-service Scenarios
Service
DB
Service
DB
Service
DB
Transaction
Hybrid Scenarios
Service
DB1
DB2
Service
DB1
DB2
Service
DB1
DB2
Transaction

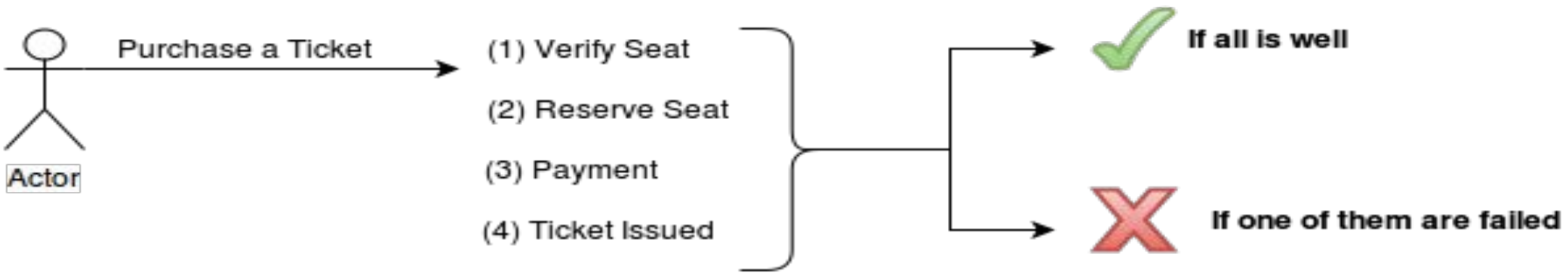
Transaction Management in JDBC :

Here are the most important points about Transaction Management in JDBC.

- A transaction means, it is a group of operations used to perform a task.
- A transaction can reach either success state or failure state.
- If all operations are completed successfully then the transaction becomes success.
- If any one of the operation fail then all remaining operations will be cancelled and finally transaction will reach to fail state.







ACID TRANSACTION GUARANTEES
Properties

All or nothing **Atomicity**

Write-ahead log

Consistency

Data always in valid state

Constraints

Isolation

Visibility of concurrent actions

Locking

Durability

Changes become permanent

Write-ahead log

ACID Transactions

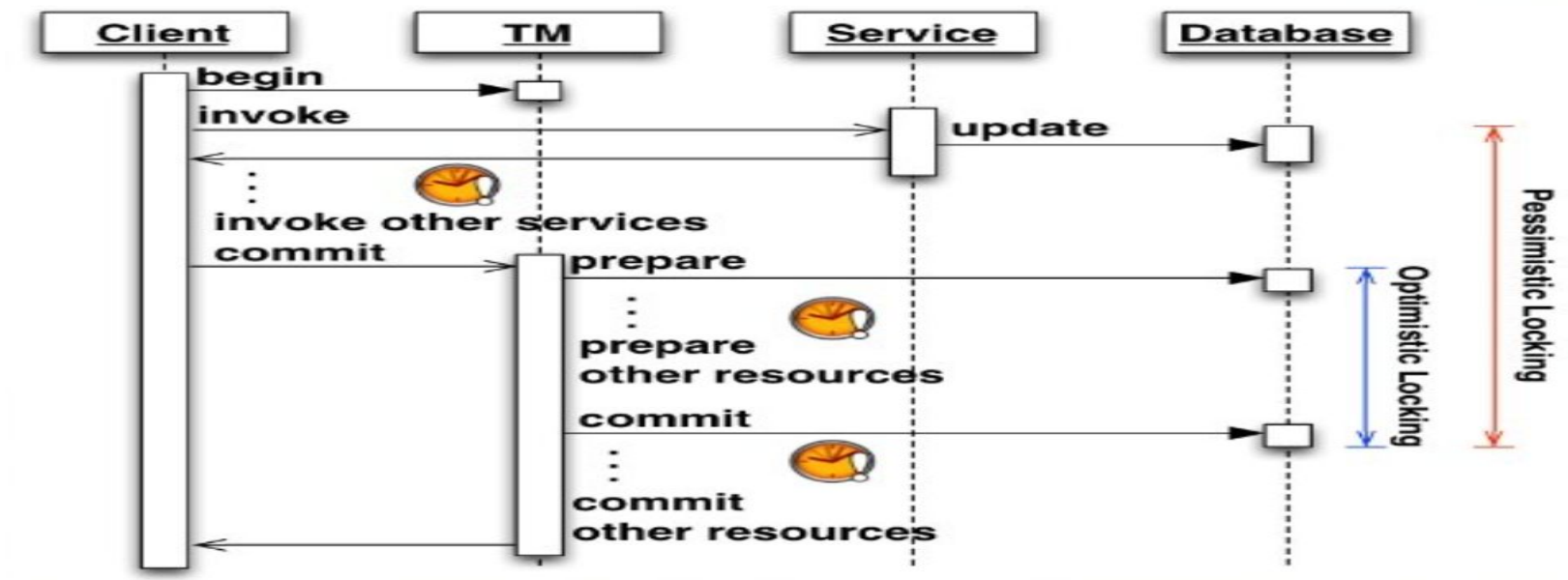
Positives

- Strong guarantees
- Tolerate (certain) failures
- Easy API to develop against

Negatives

- Blocking
- Can reduce throughput
- Requires two-phase aware resources
- Tight-coupling

Isolation: ACID



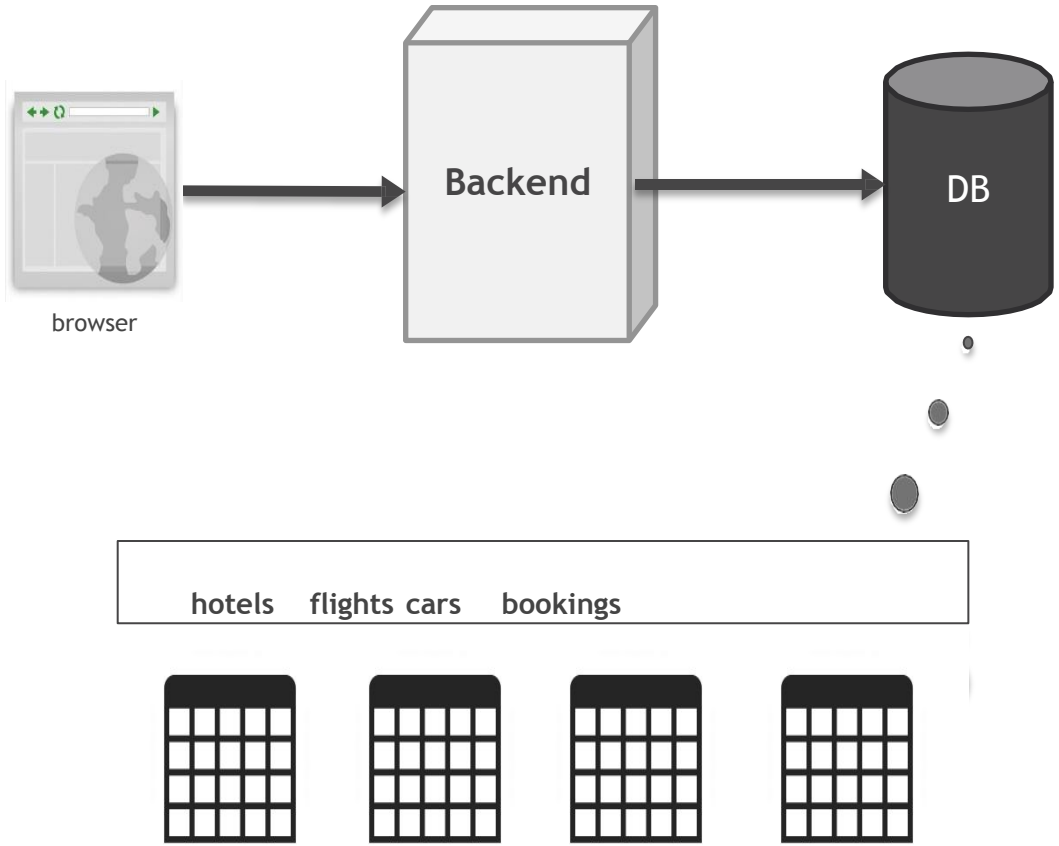
Transactions guarantee the data to always be in valid state


EXAMPLE
TRAVEL
BOOKING

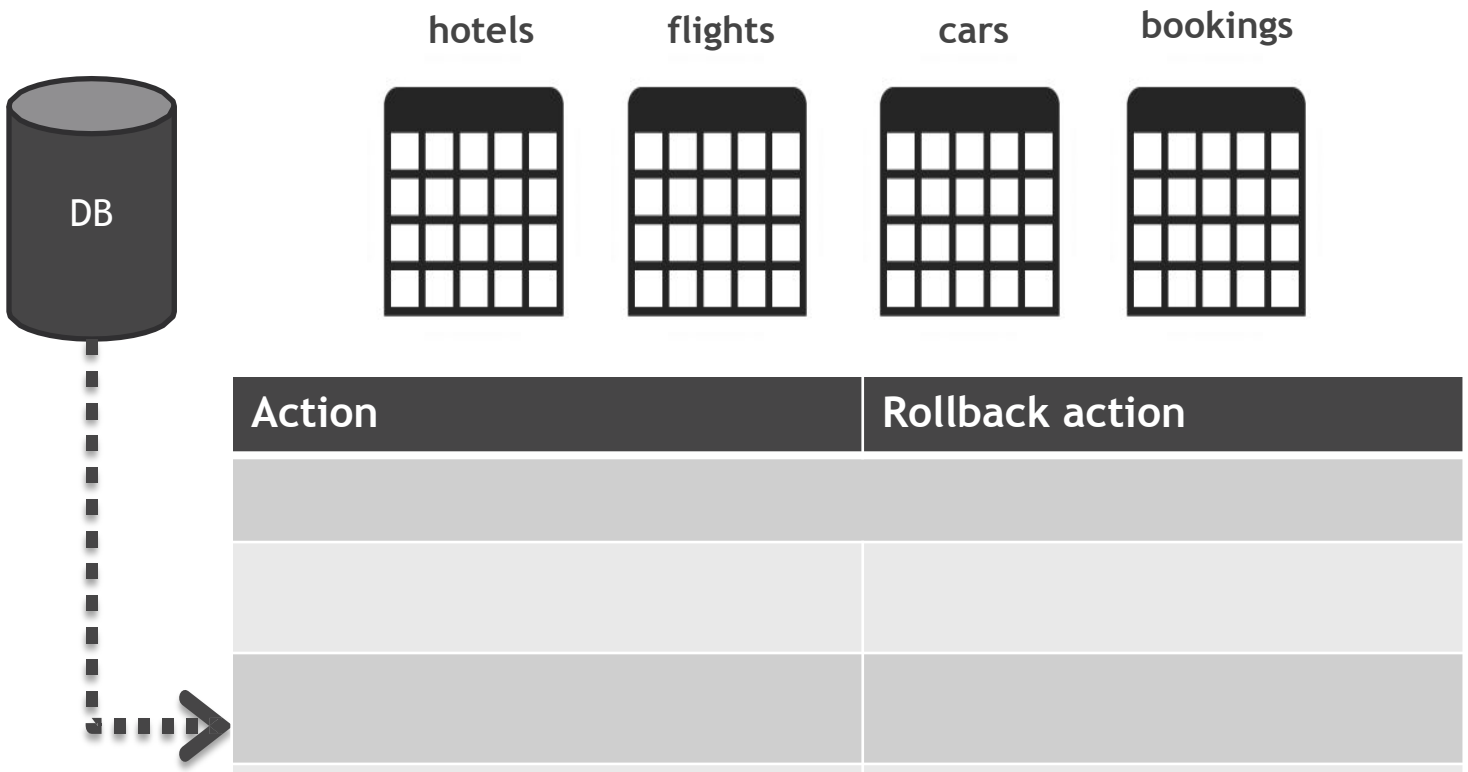
Book trip

1. Book hotel
2. Book flight
3. Hire rental car
4. Record booking

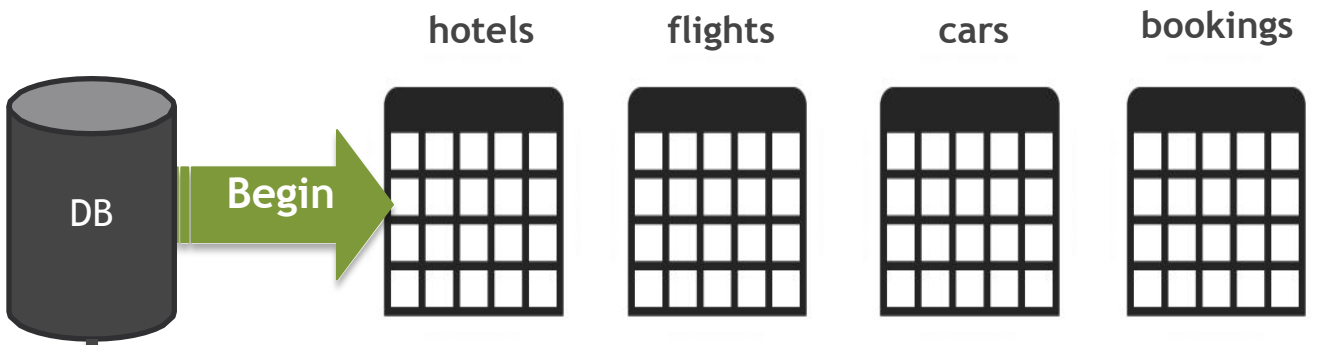
TRAVEL BOOKING SYSTEM



TRAVEL BOOKING TRANSACTION

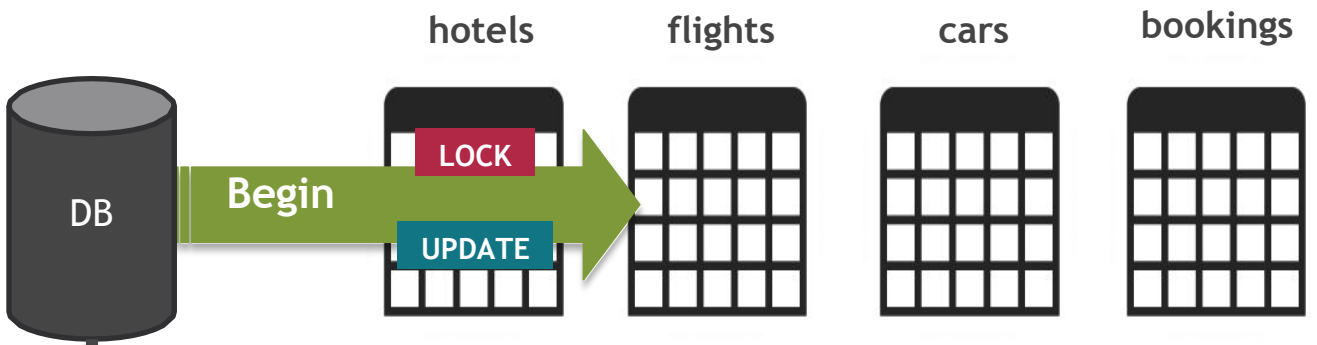


TRAVEL BOOKING TRANSACTION



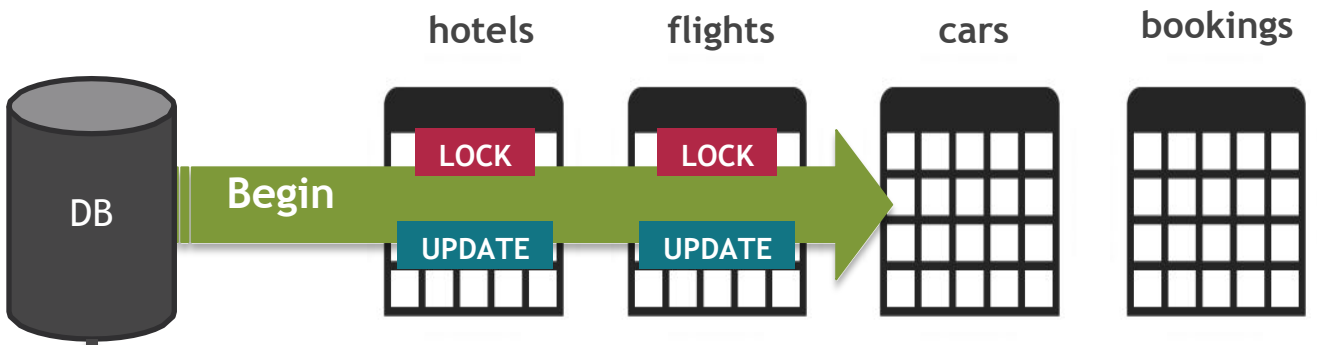
Action	Rollback action
`begin`	

TRAVEL BOOKING TRANSACTION



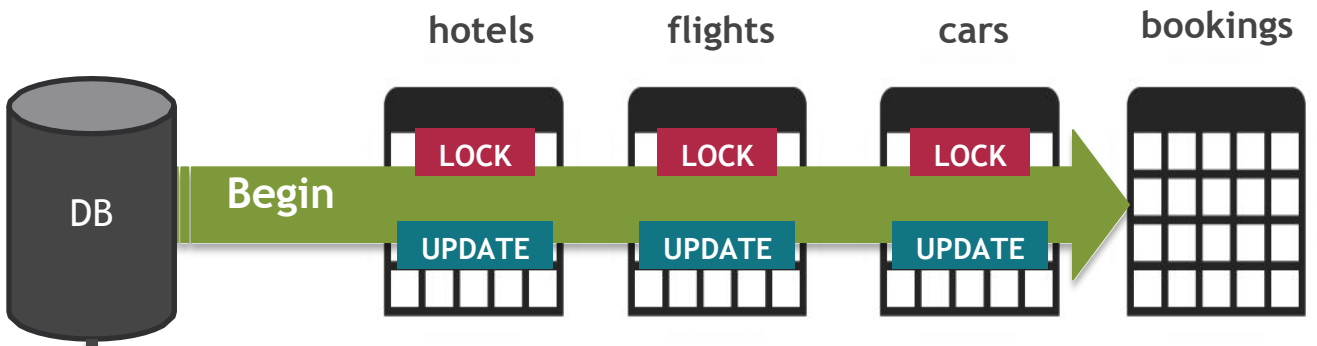
Action	Rollback action
`begin`	
`room_id=15 status=booked, traveler=Kostia`	`room_id=15 status=free, traveler=null`

TRAVEL BOOKING TRANSACTION



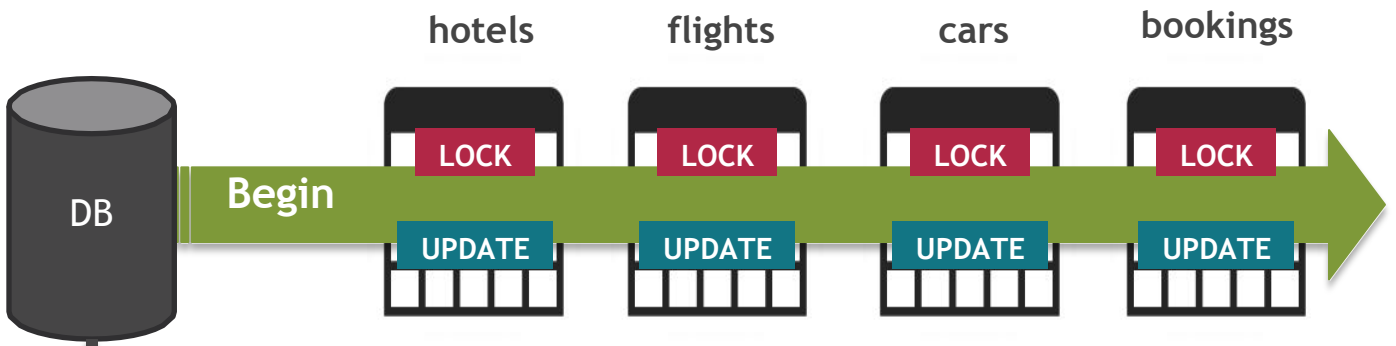
Action	Rollback action
begin	
room_id=15 status=booked, traveler=Kostia	room_id=15 status=free, traveler=null
seat_id=17c status=booked, traveler=Kostia	seat_id=17c status=free, traveler=null

TRAVEL BOOKING TRANSACTION



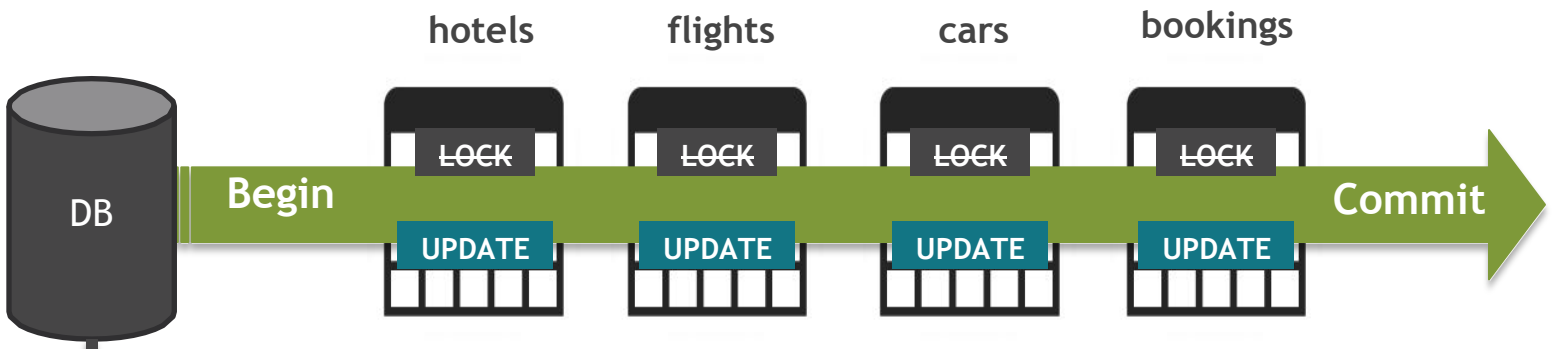
Action	Rollback action
begin	
room_id=15 status=booked, traveler=Kostia	room_id=15 status=free, traveler=null
seat_id=17c status=booked, traveler=Kostia	seat_id=17c status=free, traveler=null
car_num=1234AA7 status=booked, traveler=Kostia	car_num=1234AA7 status=free, traveler=null

TRAVEL BOOKING TRANSACTION



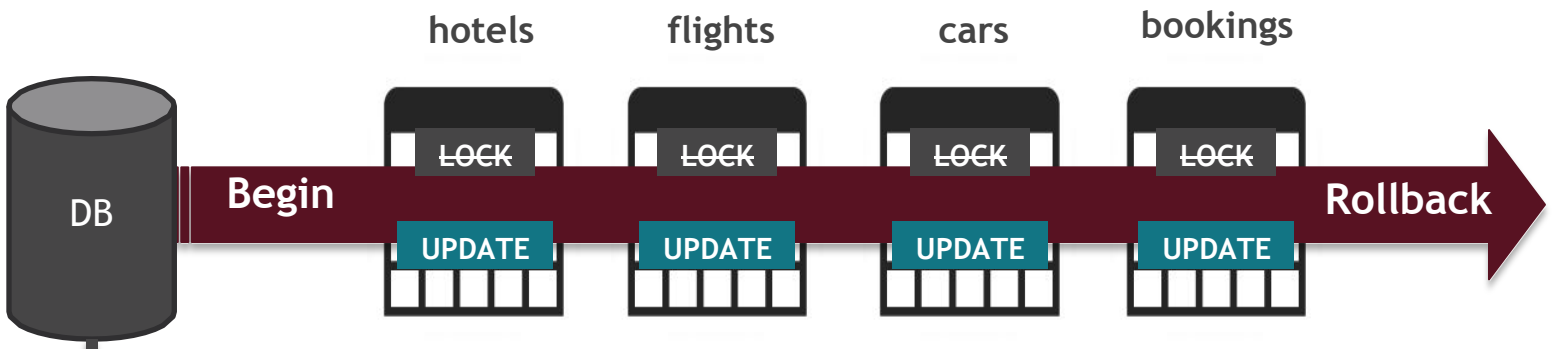
Action	Rollback action
begin	
room_id=15 status=booked, traveler=Kostia	room_id=15 status=free, traveler=null
seat_id=17c status=booked, traveler=Kostia	seat_id=17c status=free, traveler=null
car_num=1234AA7 status=booked, traveler=Kostia	car_num=1234AA7 status=free, traveler=null
INSERT traveler=Kostia, room_id=15, seat_id=17c, car_num=1234AA7	DELETE traveler=Kostia, room_id=15, seat_id=17c, car_num=1234AA7

TRAVEL BOOKING TRANSACTION



Action	~~Rollback action~~
begin	
room_id=15 status=booked, traveler=Kostia	~~room_id=15~~ ~~status=free, traveler=null~~
seat_id=17c status=booked, traveler=Kostia	~~seat_id=17c~~ ~~status=free, traveler=null~~
car_num=1234AA7 status=booked, traveler=Kostia	~~car_num=1234AA7~~ ~~status=free,~~ ~~traveler=null~~
INSERT traveler=Kostia, room_id=15, seat_id=17c, car_num=1234AA7	~~DELETE traveler=Kostia,~~ ~~room_id=15, seat_id=17c,~~ ~~car_num=1234AA7~~
commit	

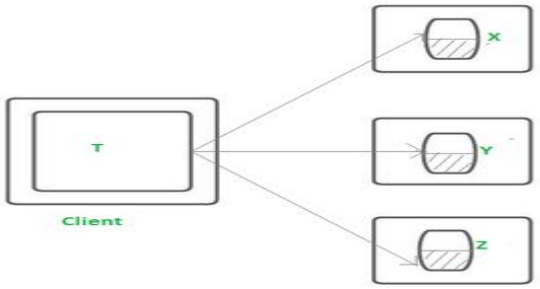
TRAVEL BOOKING TRANSACTION

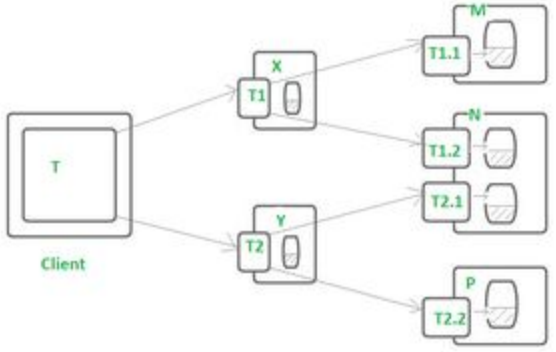


~~Action~~	Rollback action
begin	
~~room_id=15 status=booked, traveler=Kostia~~	room_id=15 status=free, traveler=null
~~seat_id=17c status=booked, traveler=Kostia~~	seat_id=17c status=free, traveler=null
~~car_num=1234AA7 status=booked, traveler=Kostia~~	car_num=1234AA7 status=free, traveler=null
~~INSERT traveler=Kostia, room_id=15, seat_id=17c, car_num=1234AA7~~	DELETE traveler=Kostia, room_id=15, seat_id=17c, car_num=1234AA7
rollback	

Types of Transaction

- Duration
  - Short Lived Transactions
  - Long Lived Transactions
- No. of Hosts
  - Single (Atomic / Local) Transactions
  - Distributed (Multiple) Transactions
- Context
  - Software system Transactions
  - Real world Business transactions
- Levels
  - Flat Transactions
  - Nested Transactions
- Enterprise
  - Intra – Enterprise Transactions
  - Inter-Enterprise Transactions





Long Term Transactions

- Transactions have a Longer Life Cycle
- Change in Business Scenario
- Not possible to use Locking
  - Long Period
  - Multiple independent Parties involved
- No Rollback features
  - Compensation Based recovery
- Durability and Consistency to be ensured
- Scenarios
  - Lot of processing like DB queries
  - Batch processes
  - Pseudo Asynchronous tasks

Short Lived vs. Long Lived Transactions	Short Lived vs. Long Lived Transactions
Shorter Life-cycle	Longer Life-cycle
Locking is Possible	Locking is Not possible
Commit / Roll-back based Mechanism	Commit / Compensation based Mechanism
Convention AICD Properties	Different C and D must be ensured
Consideration Business is NOT Required	Consideration Business is very nuch Required
Small size	Large size

Long Duration Transactions

Traditional concurrency control techniques do not work well when user interaction is required:

- **Long duration**: Design edit sessions are very long
- **Exposure of uncommitted data**: E.g., partial update to a design
- **Subtasks**: support partial rollback
- **Recoverability**: on crash state should be restored even for yet-to-be committed data, so user work is not lost.
- **Performance**: fast response time is essential so user time is not wasted.

Long-Duration Transactions

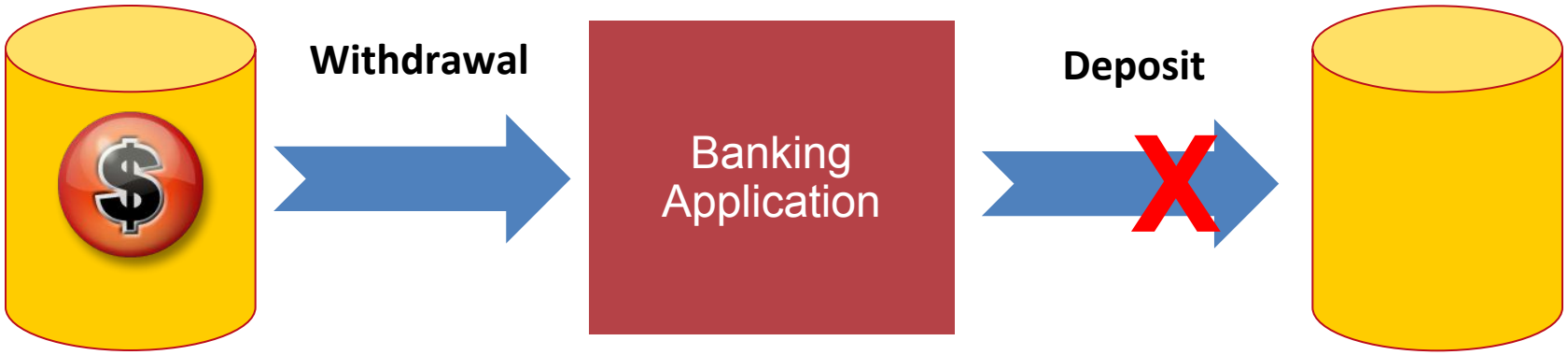
- Represent as a nested transaction
  - atomic database operations (read/write) at a lowest level.
- If transaction fails, only active short-duration transactions abort.
- Active long-duration transactions resume once any short duration transactions have recovered.
- The efficient management of long-duration waits, and the possibility of aborts.
- Need alternatives to waits and aborts; alternative techniques must ensure correctness without requiring serializability.

Definition : Distributed Transaction

“A *distributed transaction* is a single transaction that updates two or more Hosts or databases.”

OpenEdge Data Management: Database Admin

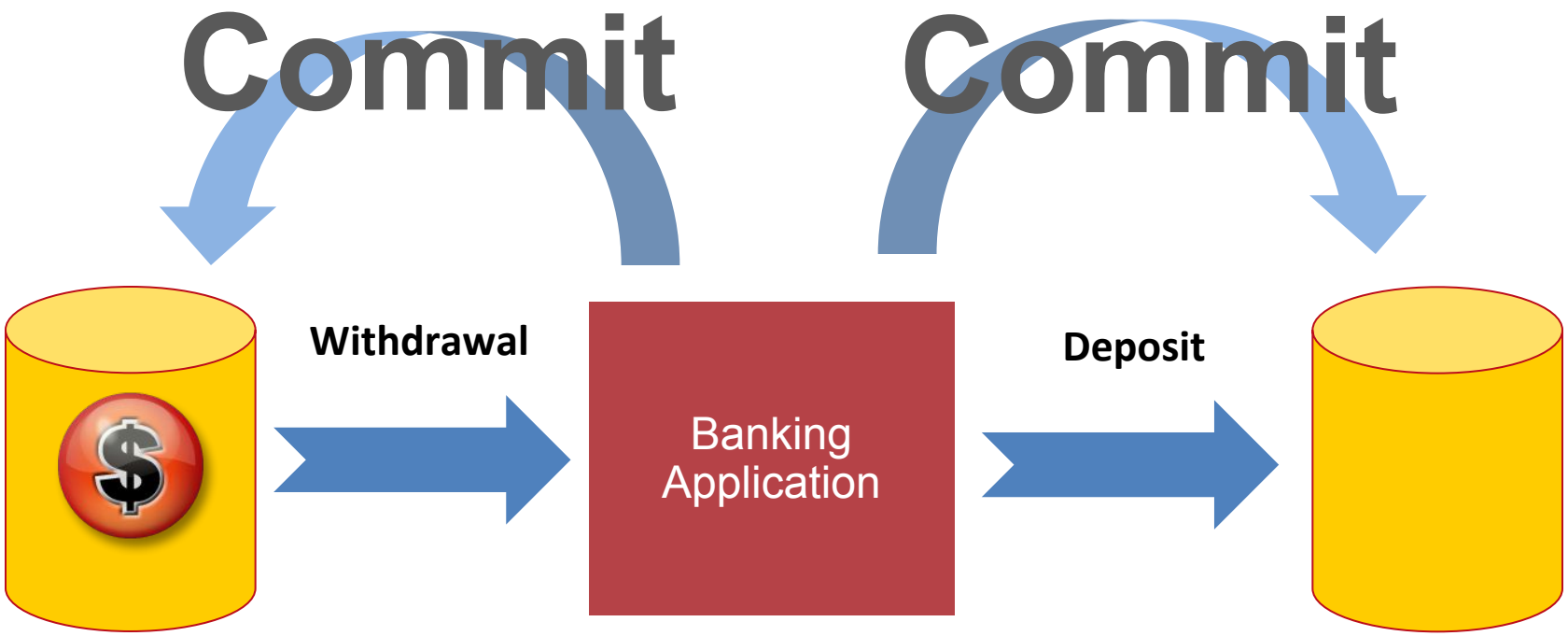
Distributed Transaction



Distributed Transaction: 2 Phase Commit

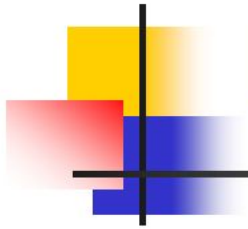
- Ensures transaction consistency across all databases
  - All commit or none
  - Commit in 2 Phases
  - Complete or Roll Back

Distributed Transaction (2PC)



Transactions

- Functionality separated into Services
  - Application comprises many Services
- Services as a separate platform
- System Independent Entities



Beyond ACID Transactions

- Composed services feature in business processes, which
  - Cross administrative boundaries
  - Are complex, i.e., long-running, failure-prone, multisite, with subtle consistency requirements
  - Cooperate with other processes, involving computational and human tasks
  - Respect autonomy and heterogeneity of components

What's needed for Transactions @Enterprise Application Context

- Business – Business Process level handling of Transactions involved multiple Enterprises
- Involved many Parties (Multi-parties)- e.g. Inter-Enterprises
- Distributed Transactions
- Long lived lasting for hours, Days, Years
- Guarantee that data is delivered and notifications sent
- No Locking for Transaction Management
- Some form of compensation for when it goes wrong
- Uniquely identify transaction across Enterprise services
- Handle errors in asynchronous services

Enterprise - Transaction Issues

- Multiple services resulting in multiple endpoints
- Loosely coupled systems
  - Maintaining Txn's only possible in closely coupled systems
- Services based on any platform
- Resources can't be kept in a *locked state*
- Alternate Recover methods

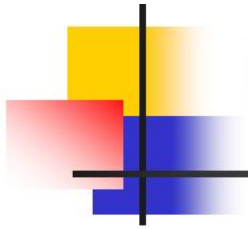
Solutions for Enterprise Application's Transaction

1. Compensation for recovery of failure
   - Open environment
   - Long duration
   - Asynchronous
2. Transaction Coordinator
   – Orchestrator for Transaction Processing
3. Protocol based e.g. BTP
   - Initiation and Transaction context
   - Termination
   - Coordination
   - Final results

Definition : Compensation

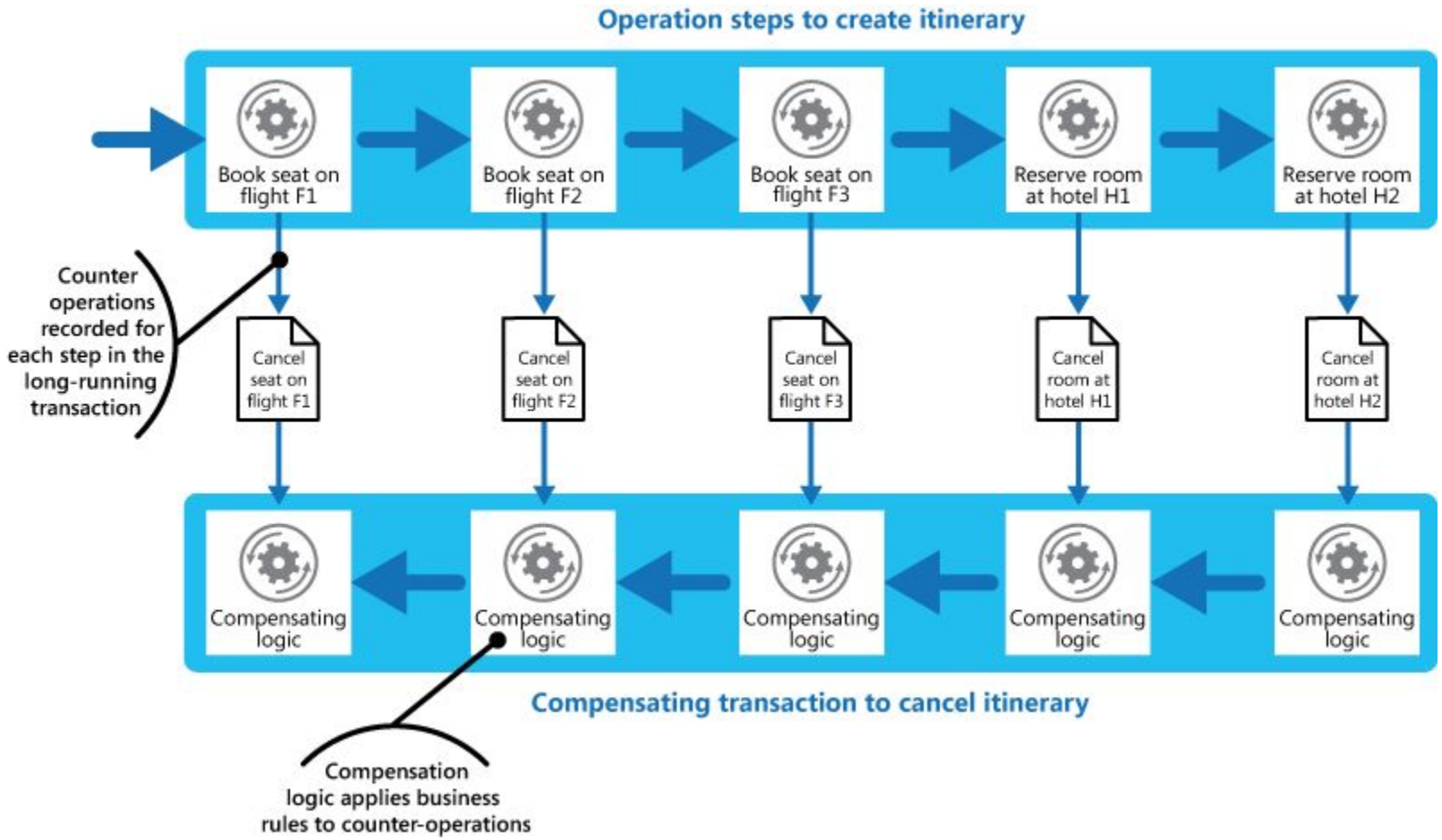
“Compensation is an action taken when something goes wrong or when there is a change of plan.”

IBM Systems Journal– April 2002

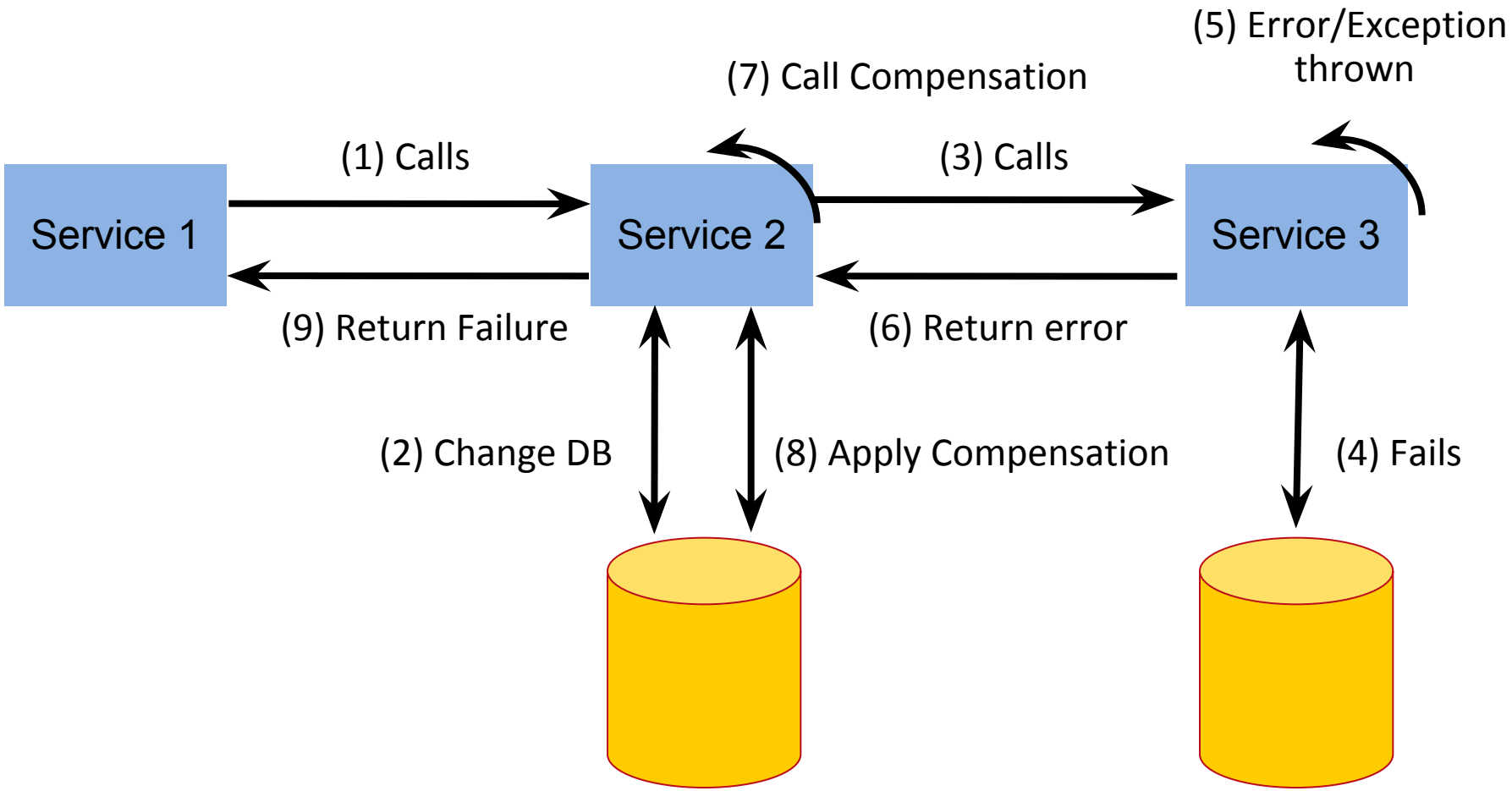


Compensation

- Something that semantically undoes the effect of a transaction
- Common in business settings
- Compensations are necessary even if imperfect
  - Deposit and withdraw
  - Reserve and cancel
  - Ship and return
  - Pay and refund


Operation steps to create itinerary
Book seat on flight F1
Book seat on flight F2
Book seat on flight F3
Reserve room at hotel H1
Reserve room at hotel H2
Counter operations recorded for each step in the long-running transaction
Cancel seat on flight F1
Cancel seat on flight F2
Cancel seat on flight F3
Cancel room at hotel H1
Cancel room at hotel H2
Compensating logic
Compensating logic
Compensating logic
Compensating logic
Compensating logic
Compensating transaction to cancel itinerary
Compensation logic applies business rules to counter-operations

Compensation

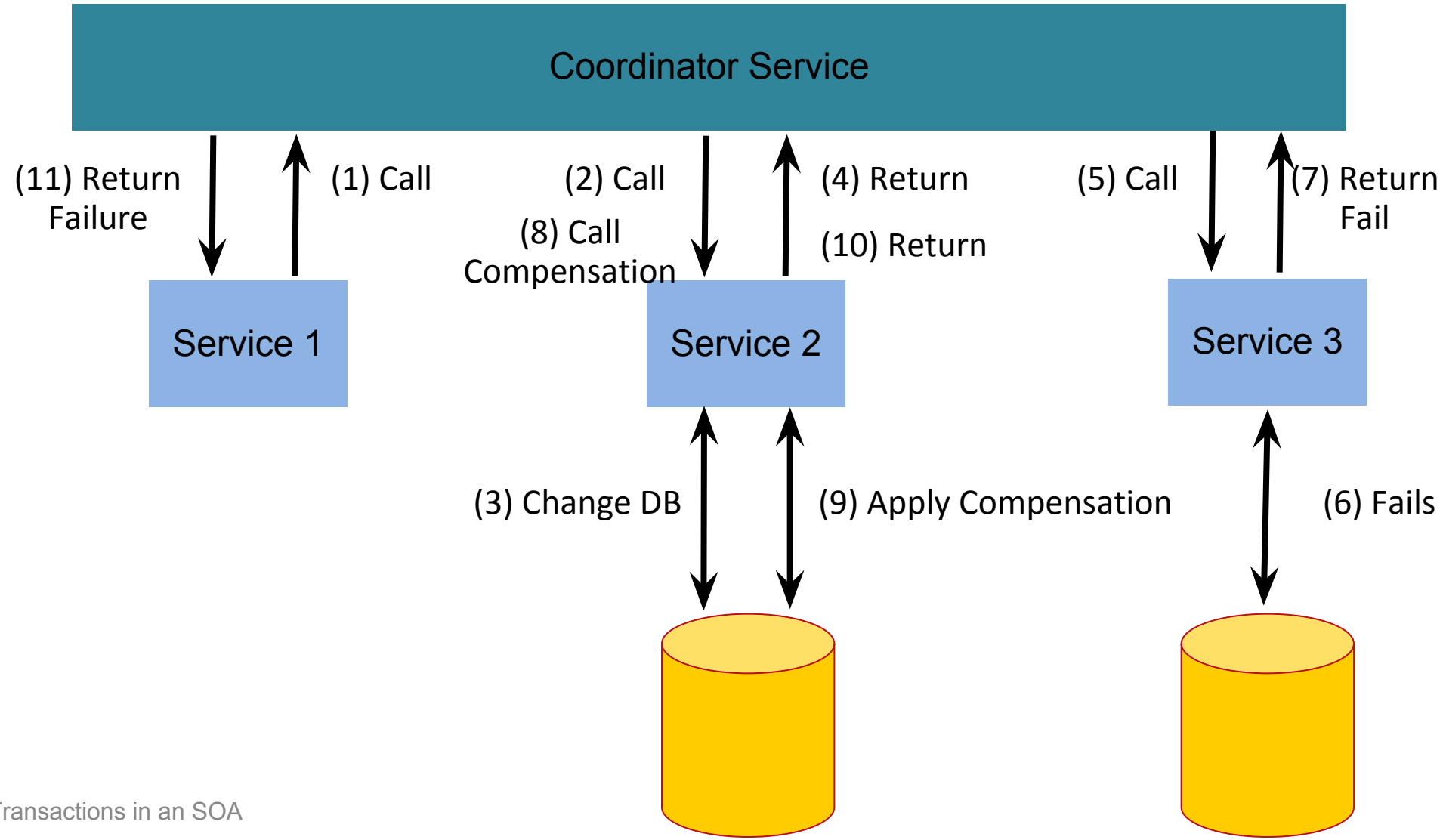


Definition : Transaction Coordination

“Orchestration of transactions through a transaction manager or process coordinator.”

SOA Systems – Feb 2007

Coordination Service



Coordination Advantages

- Fixes Asynchronous issues of Compensation
- Manages state & service information
- Central management of transaction & compensation

Two-Phase Commit (2PC)

Solution: all participating sites must agree on whether or not each action has committed.

- *Phase 1*. The sites *vote* on whether or not to commit.

  precommit: Each site prepares to commit by logging its updates before voting "yes".

- Phase 2. Commit iff all sites voted to commit.

  A central transaction *coordinator* gathers the votes.

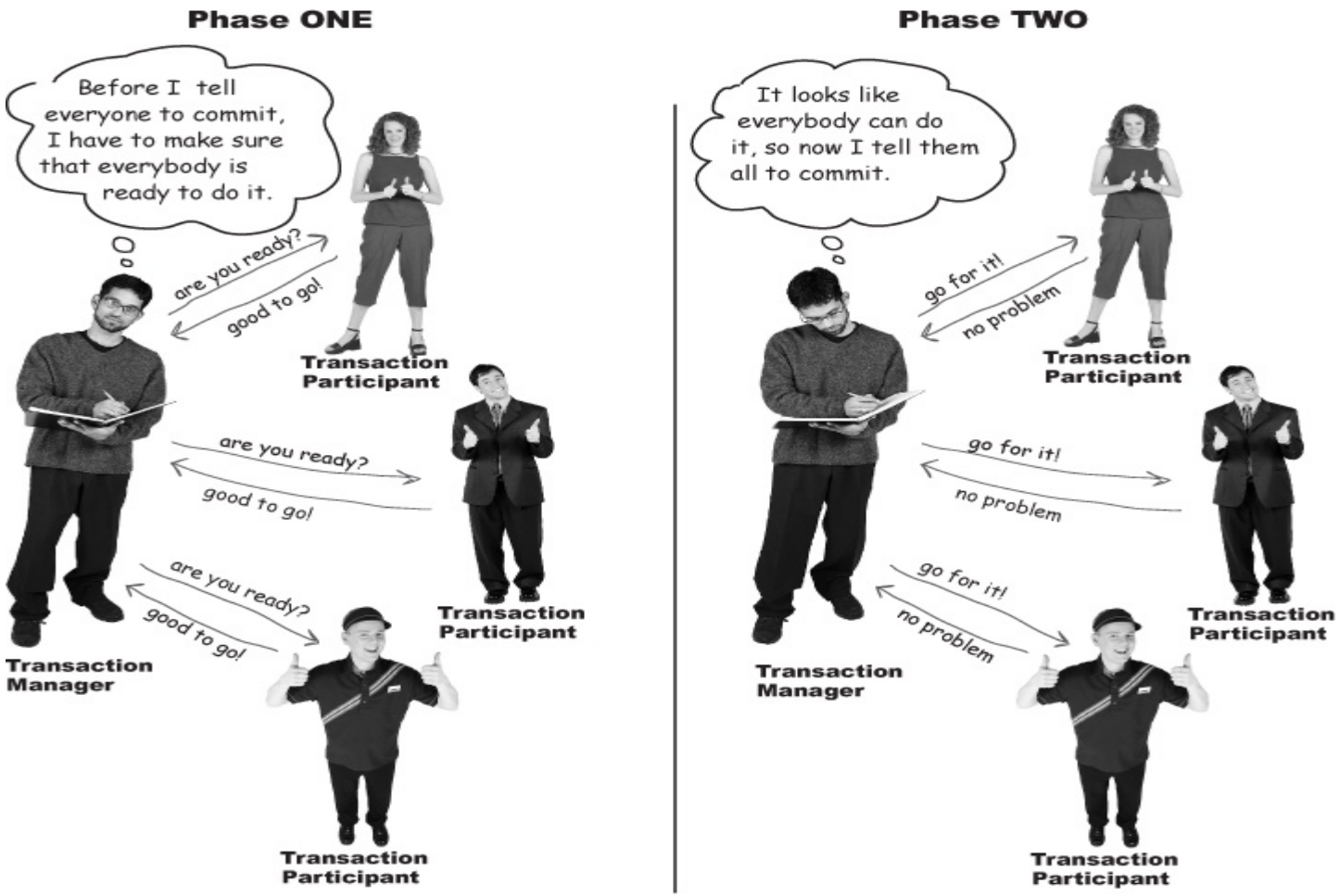
  If any site votes "no", the transaction is aborted.

  Else, coordinator writes the **commit** record to its log.

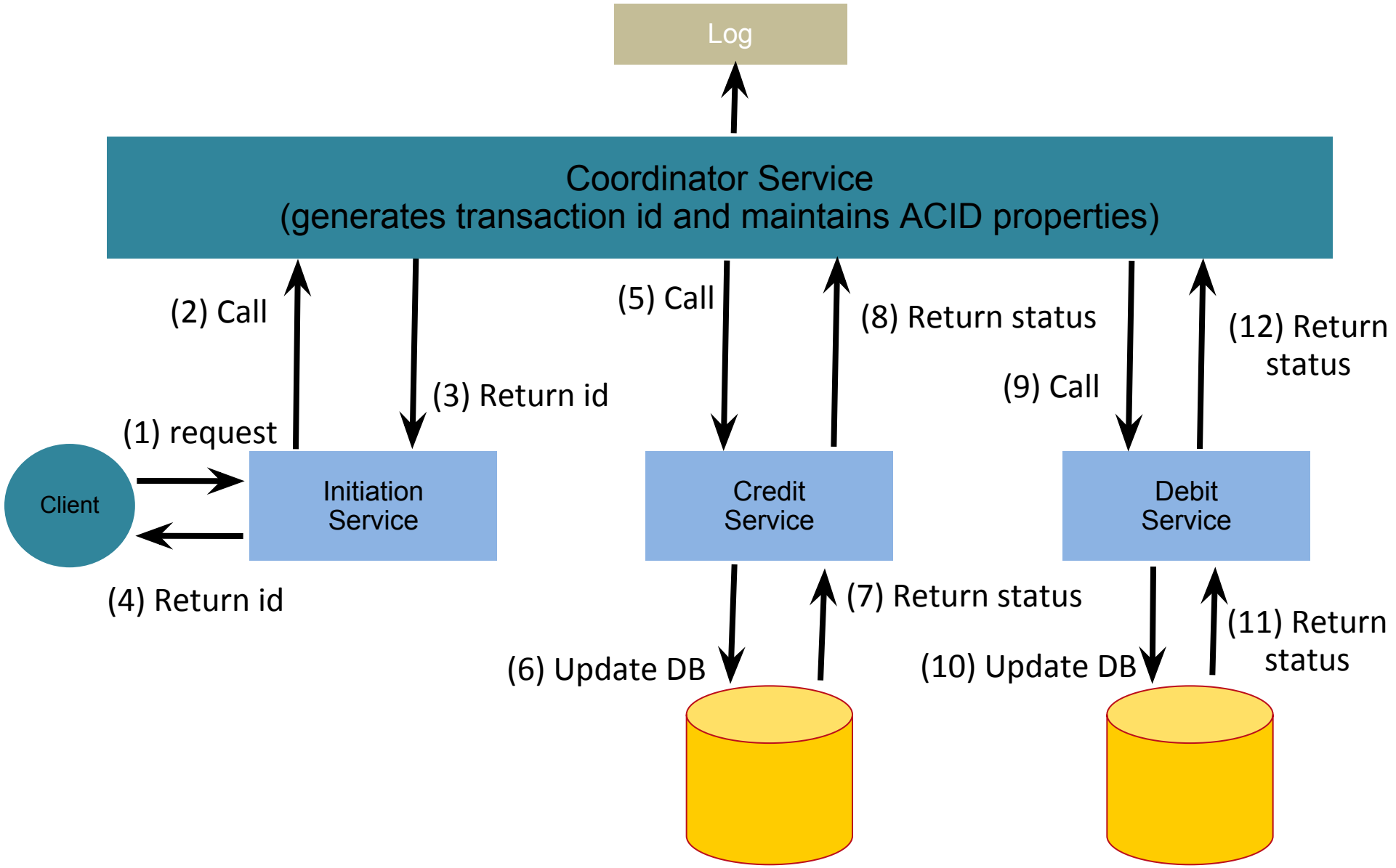
  Coordinator notifies participants of the outcome.

Note: one server ==> no 2PC is needed, even with multiple clients.

Two Phase Commit (2PC) Protocol

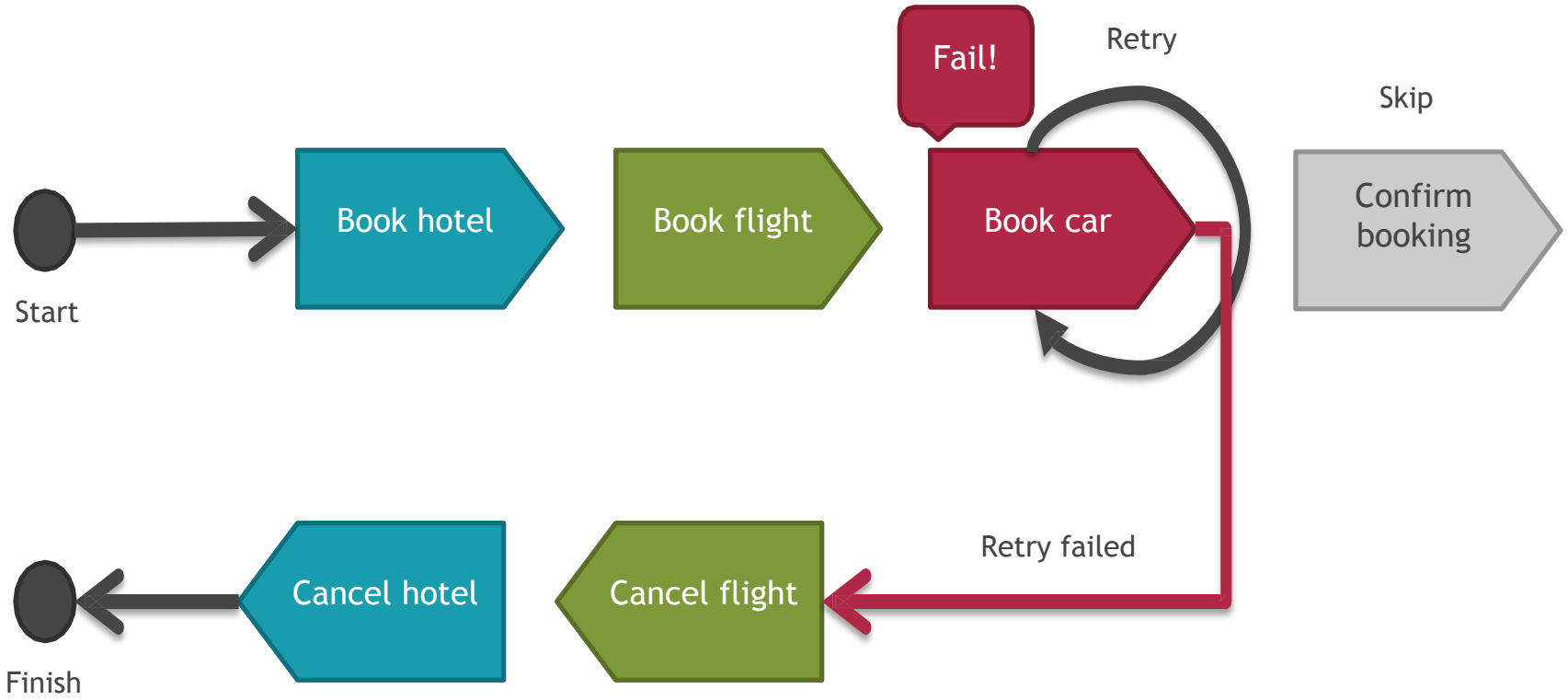


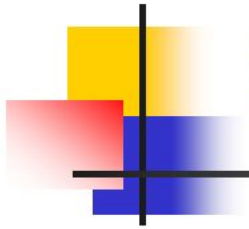
Money Transfer



WHAT IS SAGA?

SAGA EXAMPLE





Sagas

- Guarantee atomicity but not isolation
- Execute a sequence of transactions
  - If all transactions succeed, then good (**all**)
  - If any transaction fails, undo all previous in reverse order (**none**)
- Assumptions
  - Compensations succeed (eventually) for the first several members of sequence
  - Retries succeed for the last several

SAGA TYPES

Central coordinator

Routing slip (peer-to-peer)

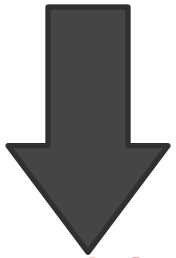
SAGA TYPES

Forward

Backward

ACID

- Availability
- Consistency
- Isolation
- Durability



2-PHASE COMMIT

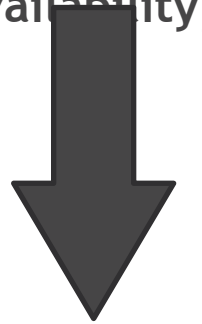
BASE

Basic Availability

Soft state

Eventual consistency

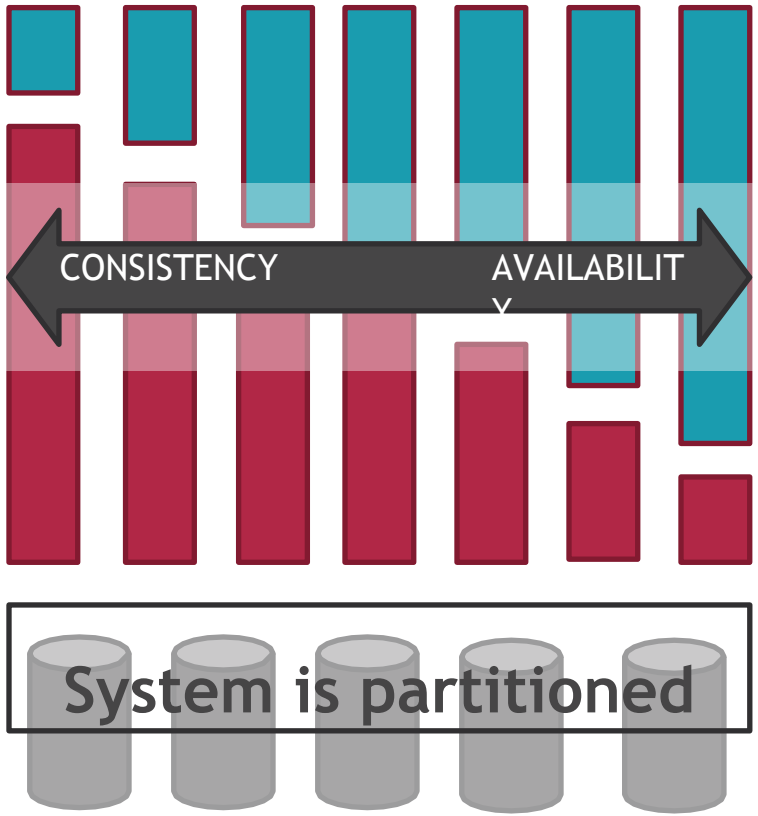
(Trading consistency for availability)



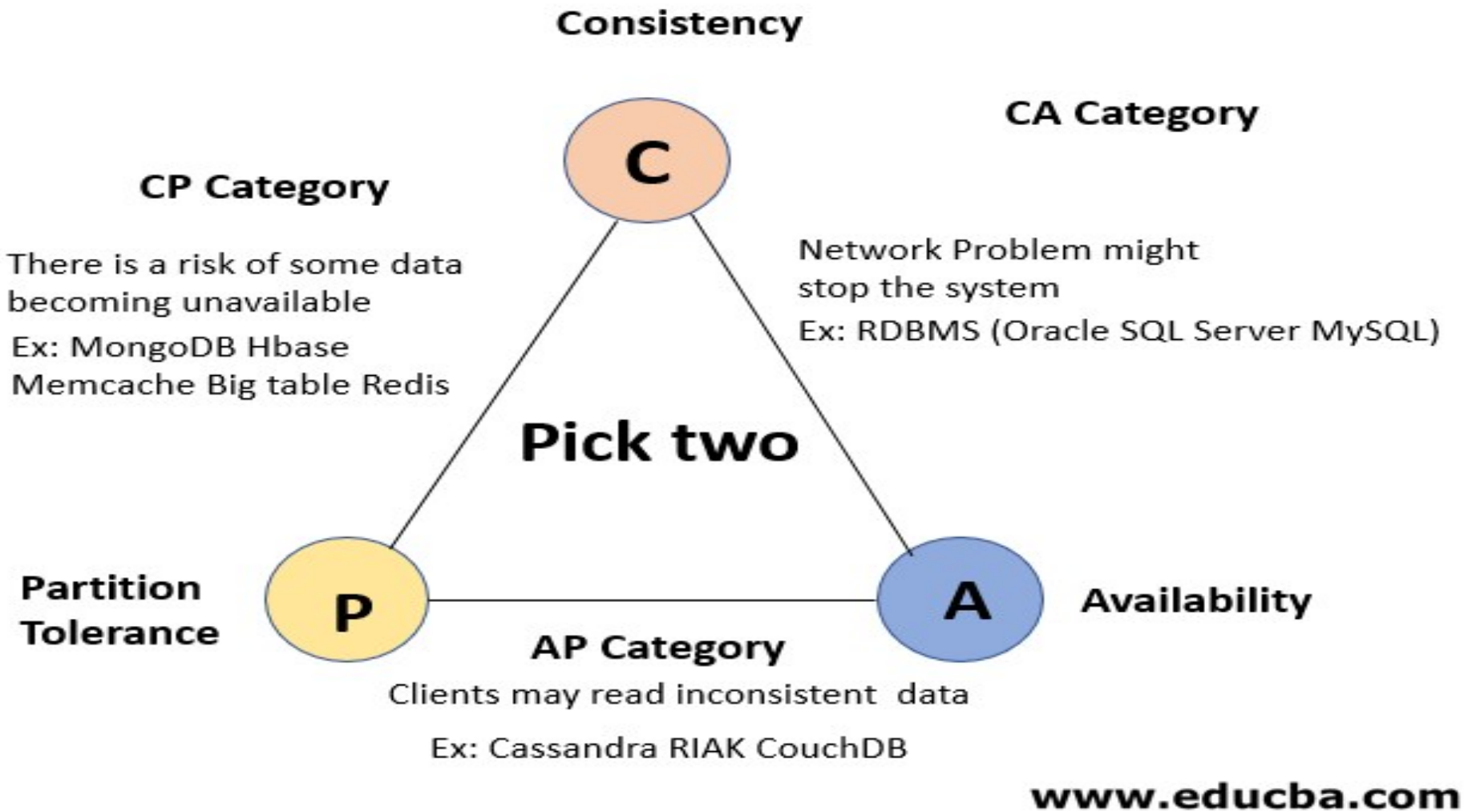
SAGA

CAP THEOREM

Consistency and Availability



CAP Theorem


Consistency
C
CA Category
CP Category
There is a risk of some data becoming unavailable
Ex: MongoDB Hbase Memcache Big table Redis
Network Problem might stop the system
Ex: RDBMS (Oracle SQL Server MySQL)
Pick two
P
A
Partition Tolerance
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra RIAK CouchDB
www.educba.com