DATA CONSISTENCY IN A HIGH VOLUME SYSTEM

Transaction Management - Revisited

- Transaction management is an important part of RDBMS oriented enterprise applications to ensure data integrity and consistency.

- **ALL [Almost]** business-related applications require a high degree of data quality.
  - Financial investment industry wastes tens of billions of dollars on failed trades
  - Lack of transaction support is a MAJOR factor leading to bad data
  - Nonexistent or poor transaction support is common

- **On the other hand**

  Transaction Management can kill performance, introduce locking issues and database concurrency problems, and add complexity to your application.

- **Ignorance** about transaction support is a significant source of the problem.

Knowledge is Power

Spring Transaction Management

Programmatic Transaction Management

- Couples application to Spring's Transaction API
- Good for small number of transactional operations
- Not good for maintenance or portability.

Declarative Transaction Management

- No dependency on the Spring Transaction API
- Least impact on application code
- Characteristics of a *Non-invasive* lightweight container

TransactionManagement Decision

Declarative Transaction Attributes

Propagation	**enum: Propagation**	**Optional propagation setting.**
Isolation	**enum: Isolation**	**Optional isolation level.**
ReadOnly	boolean	Read/write vs. read-only transaction
TimeOut	Integer [seconds]	Max Transaction time

only applies to new transactions.

```
@Service
@Transactional(propagation=Propagation.REQUIRES_NEW,
                                    isolation=Isolation.READ_COMMITTED)
public class ReadCommittedServiceImpl
                    implements edu.mum.service.ReadCommittedService {
```

Propagation Values

PROPAGATION TYPE	No Transaction	Transaction exists
MANDATORY	throw exception	use current transaction
NEVER	Run method	throw exception
NOT_SUPPORTED	Run method	Suspend current transaction, run method outside transaction
SUPPORTS	Run method	Use current transaction
REQUIRED (*default*)	**Create a transaction**	**Use current transaction**
REQUIRES_NEW	**Create a transaction**	**Suspend current transaction, create a new independent transaction**
NESTED	create a transaction	create a new nested transaction

ACID Database properties

Set of properties that guarantee that database transactions are processed reliably.

Atomicity and Durability are strict properties, i.e., Black or White, All or None

ATOMIC: The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.

CONSISTENT: A transaction transforms the database from one consistent state to another consistent state

ISOLATED**:** Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed

DURABLE: Once committed, the changes made by a transaction are persistent

Consistency and Isolation are “configurable”
&
Interdependent

Isolation in a Relational Database

- **The challenge is to maximize concurrent transactions, while maintaining consistency**
- **The shorter the lock acquisition interval, the more requests a database can process.**
- **Spring Isolation Levels:**

Spring @Transactional isolation enum reflects these values Isolation.DEFAULT – uses DB level

  - **SERIALIZABLE** (NO dirty, non-repeatable OR phantom reads)
  - **REPEATABLE READ** (NO dirty OR non-repeatable reads)
  - **READ COMMITTED** (NO dirty reads)
  - **READ UNCOMMITTED** (ANYTHING Goes)

  - ***Most databases default to READ COMMITTED***

Spring Isolation Types

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

Transaction Read Phenomena

- The ANSI/ISO standard SQL 92 refers to three different ***read phenomena*** when Transaction **A** reads data that Transaction **B** might have changed.
- **Dirty Read**

  Transaction **B** is allowed to read data from a row that has been modified by Transaction **A** and not yet committed

  If Transaction **A** rolls back, Transaction **B** has "bad" data.

Non-repeatable read

Transaction **A** reads a row twice. Transaction **B** modifies the row between reads.Transaction **A** gets inconsistent data

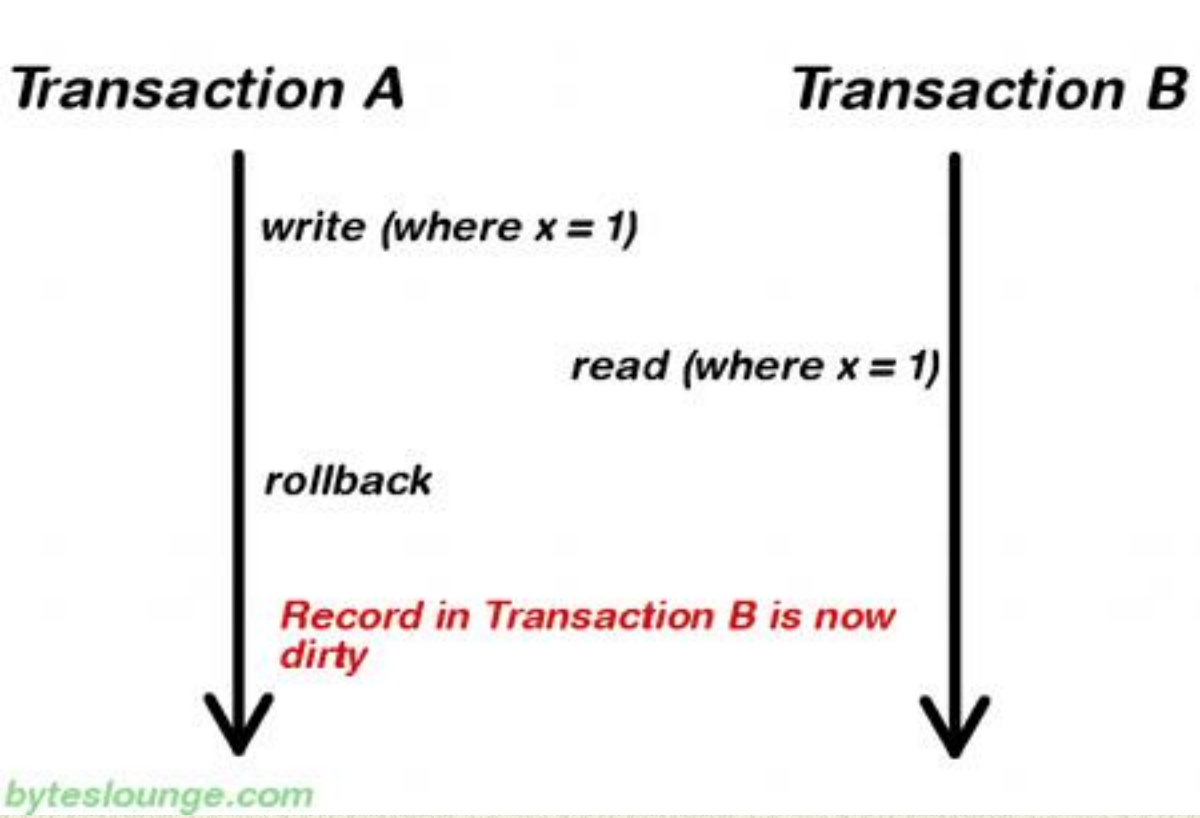
Phantom read

Transaction **A** fetches a collection twice. Transaction **B** modifies the collection between fetches. Transaction A gets inconsistent data

See Demos

DIRTY READ

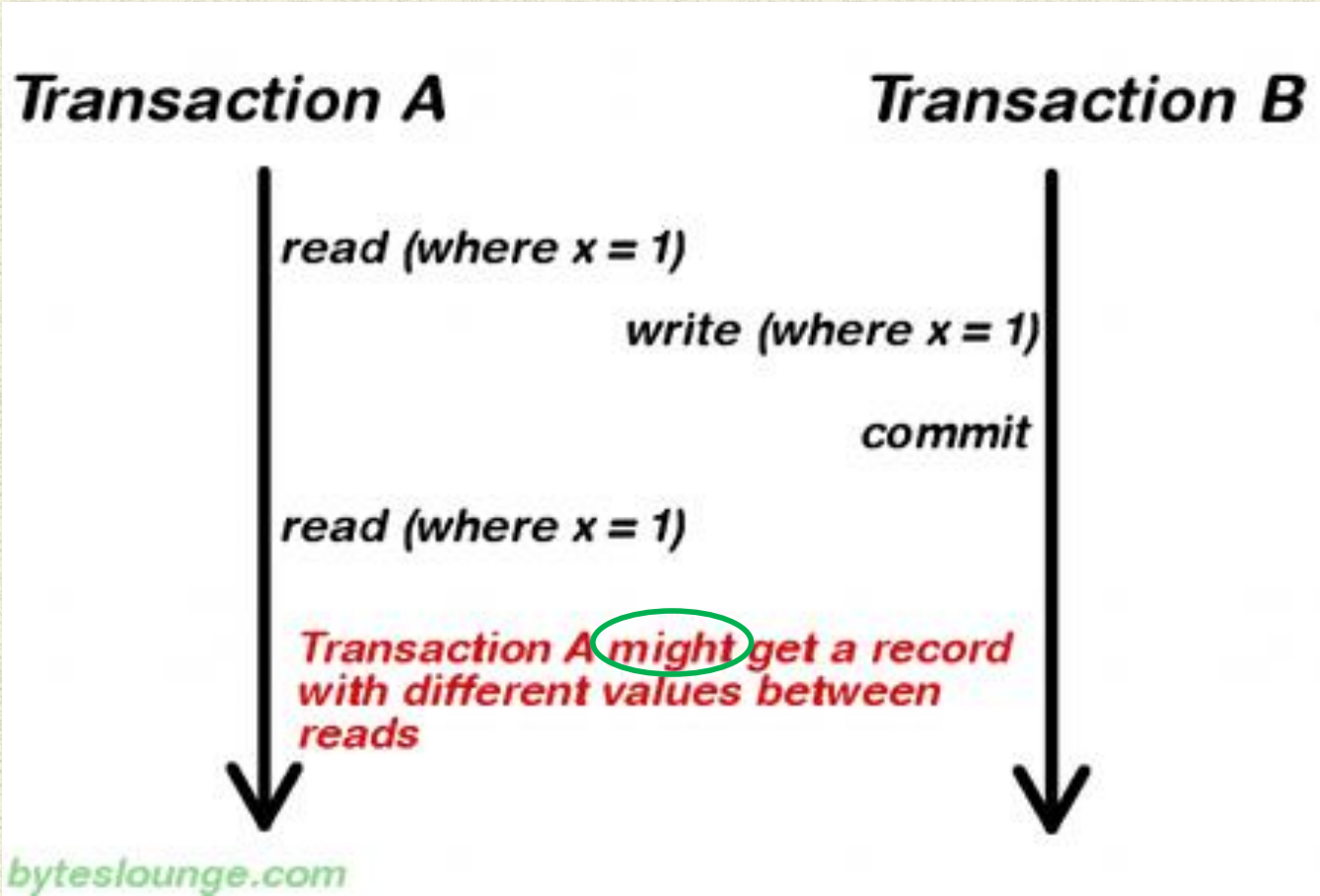
Transaction **B** is allowed to read data from a row that has been modified by Transaction **A** and not yet committed
NOTE: "x" is entity Primary key



Non Repeatable Read

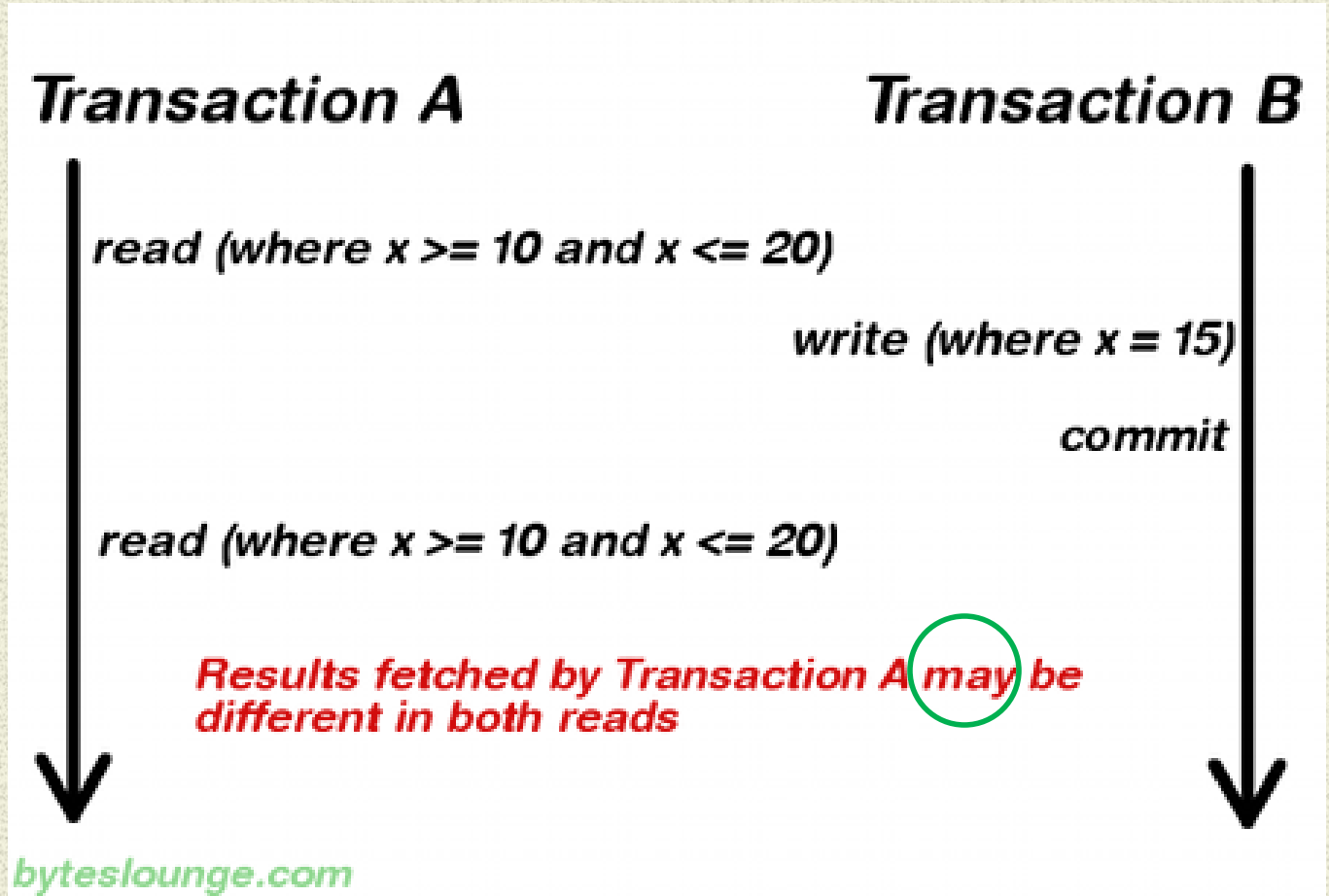
Transaction **A** reads a row twice. Transaction **B** modifies the row between reads. Transaction A gets inconsistent data

NOTE: "x" is entity Primary key



Phantom Read

Transaction **A** fetches a collection twice. Transaction **B** modifies the collection between fetches. Transaction A gets inconsistent data



Lock Mode for Data Consistency

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Locking strategies

- **Optimistic Lock**
- **Concurrent transactions can complete without affecting each other,**
- **Transactions do not need to lock the data resources that they affect.**

- **Pessimistic Lock**
- Concurrent transactions will conflict with each other
- Transactions require that data is locked when read and unlocked at commit.

Version-Based Optimistic Concurrency[Locking]

- High-volume systems
- No Connection maintained to the Database [Detached Objects]
- Effective in ***Read-Often Write-Sometimes*** scenario

- Uses ***read committed*** **isolation level**

  Gua

  1.Transaction A loads data
  2.Transaction B updates that data and commits
  3. Transaction A' updates the stale data[Throws Exception]
  In "some way" it's comparable to non-repeatable reads...
  ...By totally avoiding the second read

- Uses Anno
- **@Version**
- When entity is updated - version field is incremented.
- Used to Identify conflicts..

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that update conflicts generally don't occur
  - Keeps version #s so that it knows when they do
  - Guarantees best performance and scalability
  - The default way to deal with concurrency
  - There is no locking anywhere
  - It works well with very long conversations, including those that span multiple transactions
- First commit wins instead of last commit wins
  - An exception is thrown if a conflict would occur
    - ObjectOptimisticLockingFailureException

Optimistic Locking [Cont.]

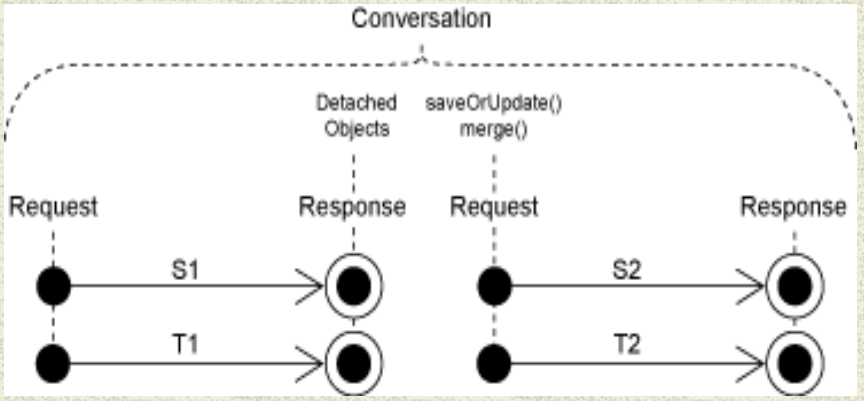
- It works well with long conversations, including those that span multiple transactions

- **LONG CONVERSATION Use Case:**

  A multi-step dialog, for example a wizard dialog interacts with the user in several request/response cycles.

- ***session-per-request-with-detached-objects***

- @Version Locking manages detached object consistency



JPA Optimistic Locking

JPA version-based concurrency control

- ***Simply add @Version to an Integer field – JPA takes care of the rest***

```
@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id = null;
   @Version
   @Column(name = "version")
   private int version = 0;
```

@Version property is incremented on every DB write

Checked for consistency on every update

```
Hibernate:
 update purchaseOrder
 set orderNumber=?,version=? [increment version ]
 where id=? and version=? [if detached version=DB version]
```

Main Point

A unit of work is a representation of a process that is only considered when all its parts are complete.

Science of Consciousness: *Wholeness is greater than the sum of the parts.*