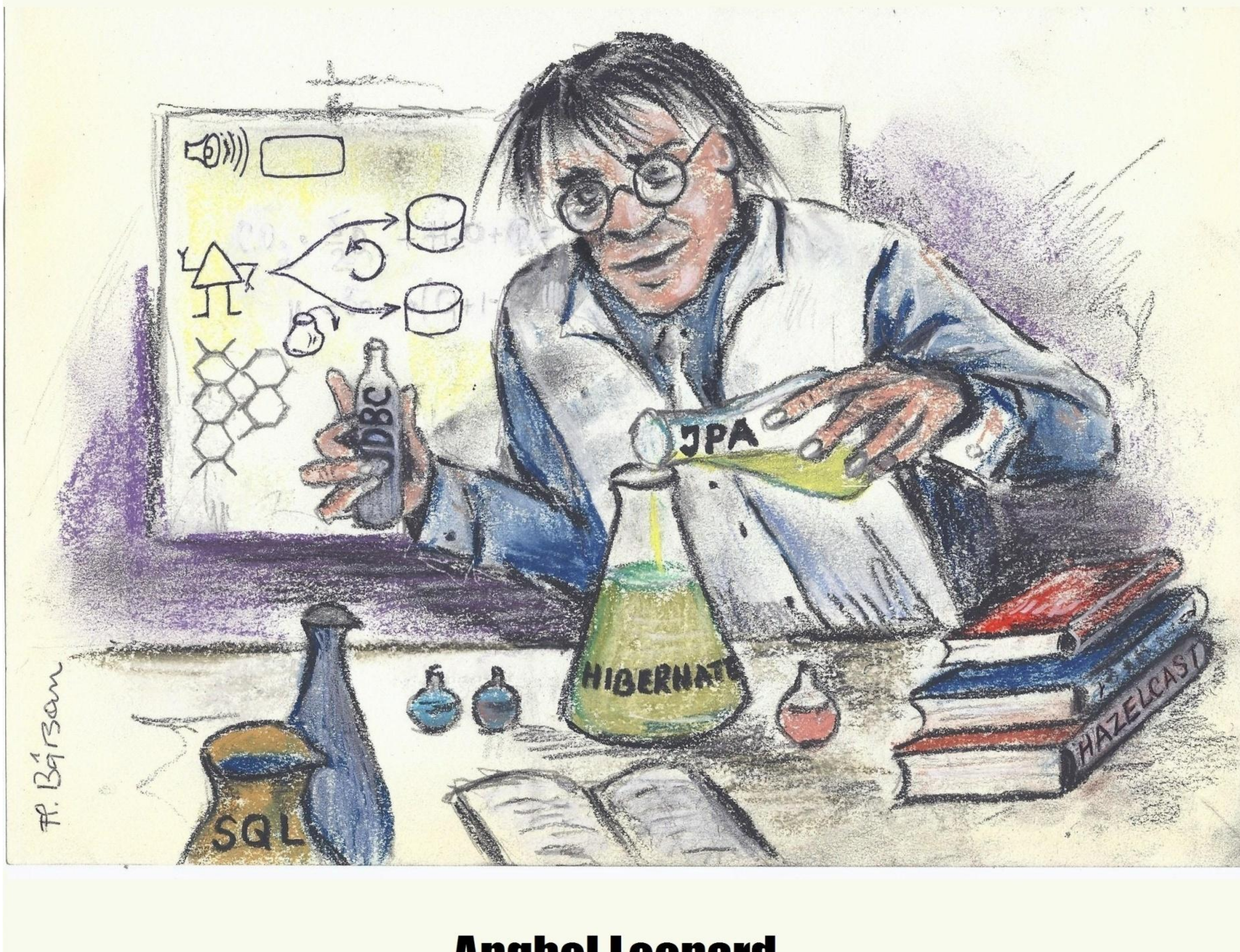
Java Persistence Performance Illustrated Guide

SQL - JDBC - JPA - Hibernate - Hazelcast



Anghel Leonard

Java Persistence Performance Illustrated

Anghel Leonard

Contents

Introduction

This book is a collection of images and headlines meant to expose the best practices of exploiting the Java persistent layer from the performance perspective. The first part is dedicated to SQL, the second part is dedicated to relational database systems, the third part (and the most consistent) is dedicated to JPA and Hibernate, and the last part is dedicated to Hazlecast in-memory data grid.

Contact: leoprivacy@yahoo.com
@anghelleonard

Enjoy it :)

Special thanks!

Special thanks to Mr. Barsan Florian for designing the cover of this book :)

Contact: bpipimus@yahoo.com

Part I

SQL

Lifecycle

SQL statement lifecycle

Avoid DDL generation in production

SQL statement lifecycle

The following illustration depicts the rough lifecycle of an SQL statement:

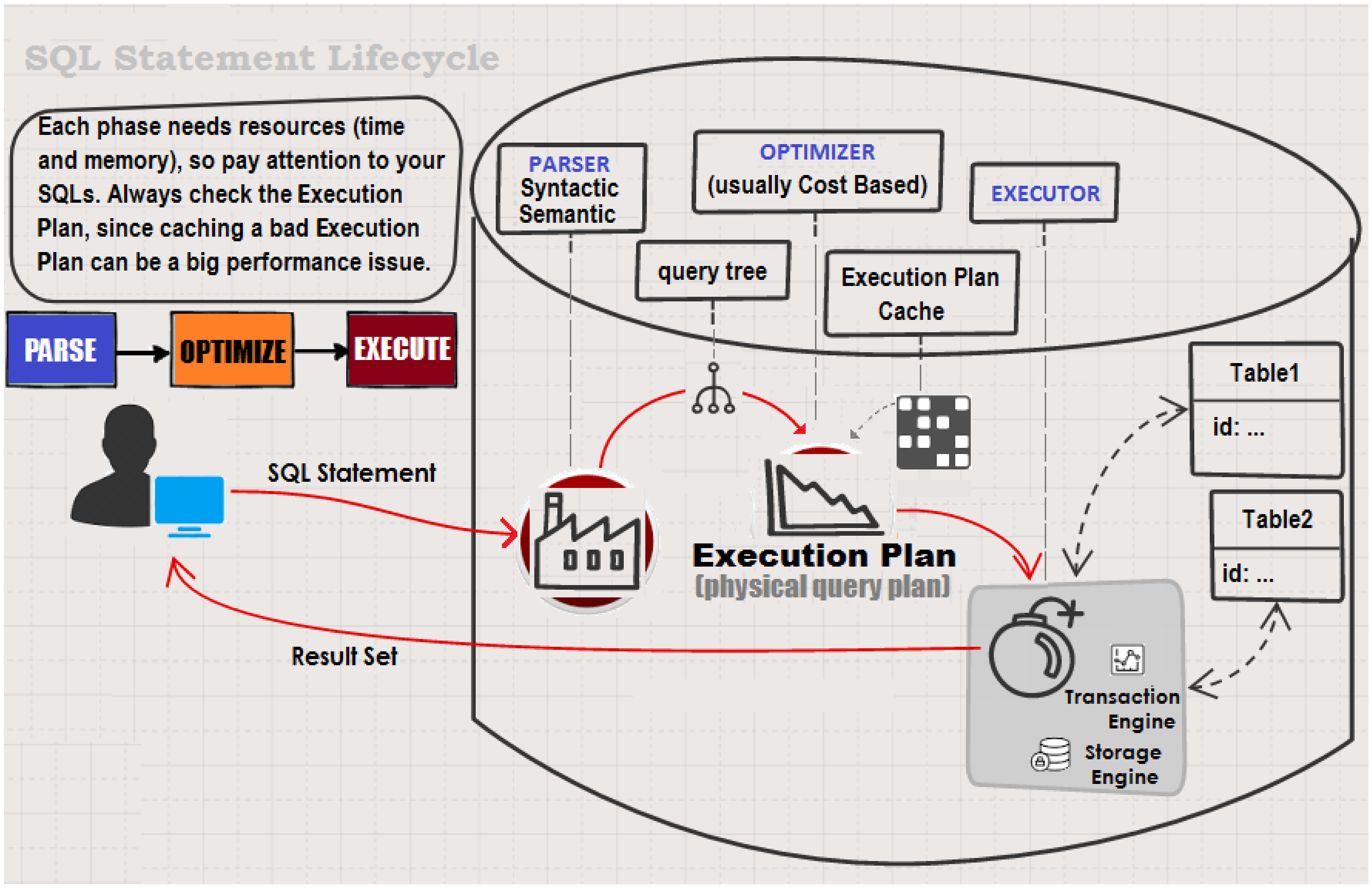
- **Each SQL statement needs to be parsed, optimized and executed, and each of these phases (actions) takes time and memory (checkout the Parser, Optimizer and Executor order of invocation)**

- **Trying to consume fewer resources will result in shorter transactions and, as a consequence, in better performance**

- **As a rule of thumb, always check the Execution Plan (e.g. pay attention if the indexes are used as expected) since most probably your database will reuse it, and reusing a bad Execution Plan is obviously a performance issue**

- **Take into consideration the settings for server-side/client-side statement caching. Commonly, these settings are specific to each relational database.**

- **Read the manual of your RDBMS and ORM in order to "harvest" any specific and useful setting that it is meant to increase the performance. But, always pay attention to the trade-off that need to be made in order to increase the performance.**


SQL Statement Lifecycle
Each phase needs resources (time and memory), so pay attention to your SQLs. Always check the Execution Plan, since caching a bad Execution Plan can be a big performance issue.
PARSE
OPTIMIZE
EXECUTE
PARSER
Syntactic
Semantic
OPTIMIZER
(usually Cost Based)
EXECUTOR
query tree
Execution Plan Cache
SQL Statement
Execution Plan
(physical query plan)
Result Set
Table1
id: ...
Table2
id: ...
Transaction Engine
Storage Engine

Avoid DDL generation in production

The following illustration depicts DDL generation:

- **In production, avoid relying on `hbm2ddl.auto` and `hibernate.ddl-auto` and any similar settings (counterparts) for generating the application DDL**

- **In production, just skip this setting and it does not perform any creation or modification automatically**

- **In production, rely on version control and robust schema evolution via Flyway (https://flywaydb.org/) or Liquibase (https://www.liquibase.org/) or similar products**

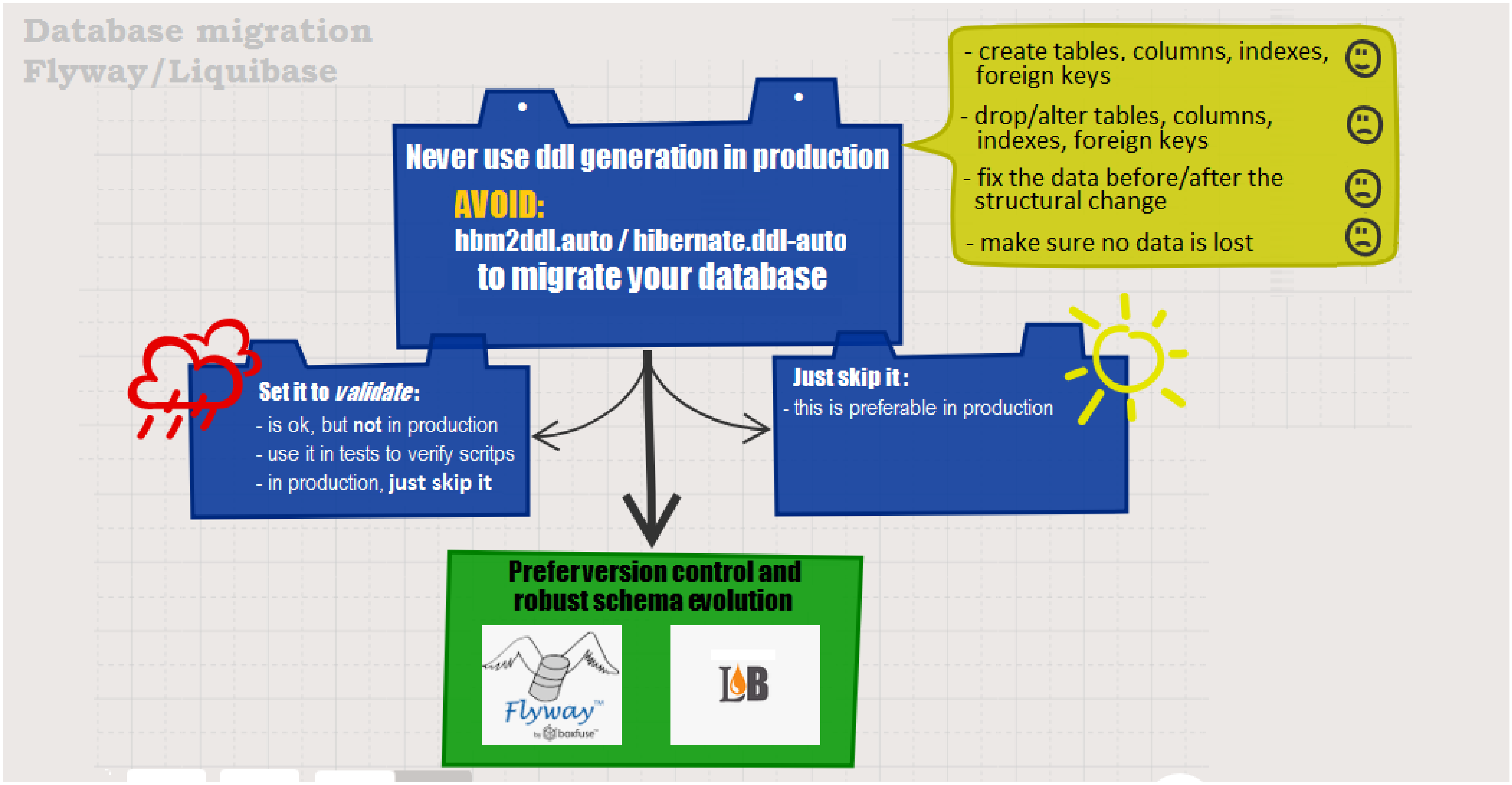
Applications:

- **How To Migrate MySQL Database Using Flyway - Database Created Via spring.flyway.schemas**
- **How To Programmatically Setup Flyway And MySQL DataSource**
- **How To Migrate MySQL Database Using Flyway - Database Created Via createDatabaseIfNotExist**
- **How To Auto-Create And Migrate Two Databases In MySQL Using Flyway**
- **How To Programmatically Setup Flyway And PostgreSQL DataSource**
- **How To Migrate PostgreSQL Database Using Flyway - Use The Default Database postgres And Schema public**
- **How To Migrate Schema Using Flyway In PostgreSQL - Use The Default Database postgres And Schema Created Via spring.flyway.schemas**
- **How To Auto-Create And Migrate Two Schemas In PostgreSQL Using Flyway**
- **How To Auto-Create And Migrate Schemas For Two Data Sources (MySQL and PostgreSQL) Using Flyway**

- **In Spring Boot, we can rely on `schema-*.sql` as in the following two examples (but, there is no version control or migration support)**

Applications:

- **How To Generate A Schema Via schema-*.sql In MySQL**
- **How To Generate Two Databases Via schema-*.sql And Match Entities To Them Via @Table In MySQL**



Database migration Flyway/Liquibase

Never use ddl generation in production

AVOID:
hbm2ddl.auto / hibernate.ddl-auto
to migrate your database

- create tables, columns, indexes, foreign keys
- drop/alter tables, columns, indexes, foreign keys
- fix the data before/after the structural change
- make sure no data is lost

Set it to *validate*:
- is ok, but **not** in production
- use it in tests to verify scritps
- in production, **just skip it**

Just skip it :
- this is preferable in production

Prefer version control and robust schema evolution

Indexes/statements

Using functions in the WHERE clause

Why the index column order matter

Primary key vs. unique key

LIKE operator vs. equal

LIKE wildcards usage

UNION vs. UNION ALL and JOIN flavors

Using functions in the WHERE clause

The following illustration depicts using SQL functions in the `WHERE` clause:

- **We should already know that the database indexes are meant to increase the performance of our SQL queries. Basically, an index can turn a slow SQL into a fast SQL and influence the choice of the Execution Plan.**

- **In order to be used, the existence of an index is necessary, but it is not sufficient. In other words, an index is used or ignored depending on how we write the SQL statement.**

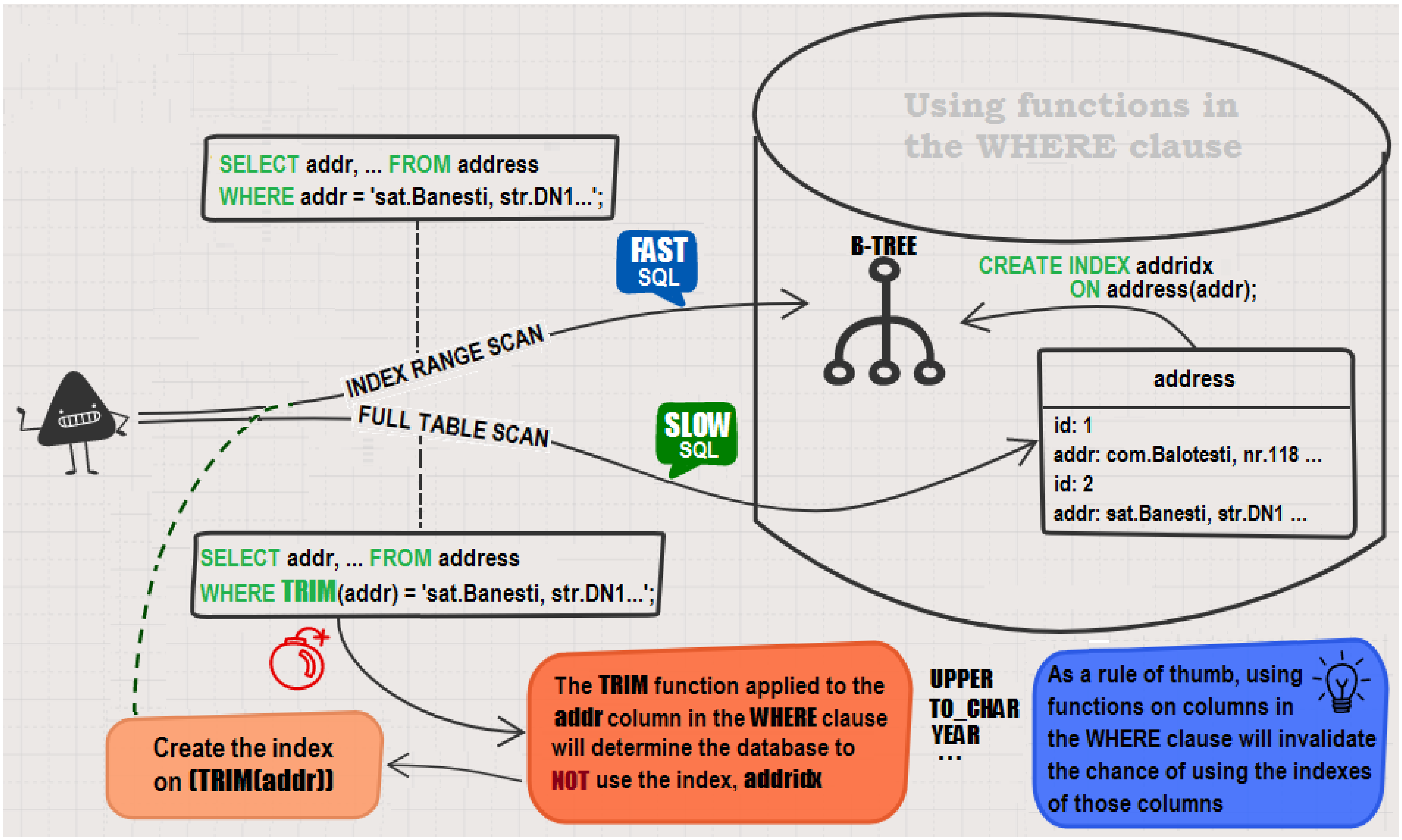
- **For example, if we have an index on a column, and we apply an SQL function in the `WHERE` clause to this column, then the index will not be used. The index is unusable because the database doesn't find any corresponding index on *function_name*(*column*). The index is on *column* only.**

- **A potential solution is to create an index on *function_name*(*column*); this is known as function-based index:**

```
CREATE INDEX addridx ON address (TRIM(addr));
```

- **Or, if function-based index is not supported:**

```
ALTER TABLE address ADD trim_addr AS TRIM(addr);
CREATE INDEX addridx ON address (trim_addr);
```


Using functions in the WHERE clause
SELECT addr, ... FROM address
WHERE addr = 'sat.Banesti, str.DN1...';
FAST SQL
INDEX RANGE SCAN
B-TREE
CREATE INDEX addridx
ON address(addr);
FULL TABLE SCAN
SLOW SQL
address
id: 1
addr: com.Balotesti, nr.118 ...
id: 2
addr: sat.Banesti, str.DN1 ...
SELECT addr, ... FROM address
WHERE TRIM(addr) = 'sat.Banesti, str.DN1...';
Create the index on (TRIM(addr))
The TRIM function applied to the addr column in the WHERE clause will determine the database to NOT use the index, addridx
UPPER
TO_CHAR
YEAR
...
As a rule of thumb, using functions on columns in the WHERE clause will invalidate the chance of using the indexes of those columns

Why the index column order matters

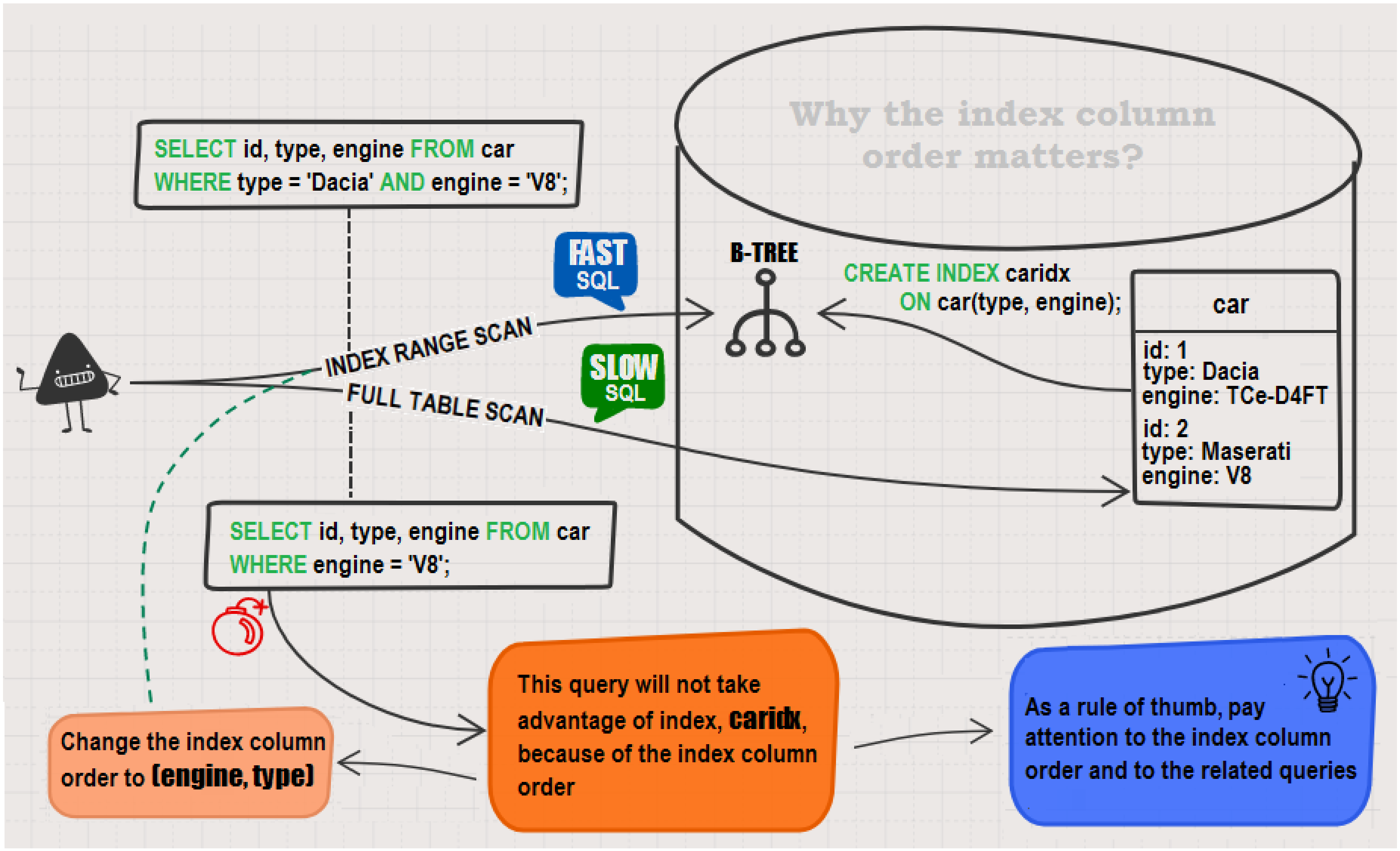
The following illustration depicts why the index column order matters:

- **A database index can be created on a single column or on multiple columns. In the latter case, the index is known as a *concatenated* index.**

- **The essence of this section is related to the *concatenated* index column order. The performance of our SQL statements may be adversely affected if we don't take into account the fact that the *concatenated* index column order has a major impact on the index usability.**

- **The below picture depicts the usability of a *concatenated* index created on two columns. Notice how the *concatenated* index column order affects the second query:**

```
SELECT id, type, engine FROM car WHERE engine = 'V8';
```


Why the index column order matters?
SELECT id, type, engine FROM car WHERE type = 'Dacia' AND engine = 'V8';
FAST SQL
INDEX RANGE SCAN
SLOW SQL
FULL TABLE SCAN
B-TREE
CREATE INDEX caridx ON car(type, engine);
car
id: 1 type: Dacia engine: TCe-D4FT
id: 2 type: Maserati engine: V8
SELECT id, type, engine FROM car WHERE engine = 'V8';
This query will not take advantage of index, caridx, because of the index column order
Change the index column order to (engine, type)
As a rule of thumb, pay attention to the index column order and to the related queries

Primary key vs. Unique key

The following two illustration depicts primary keys headlines and primary key versus unique key topic:

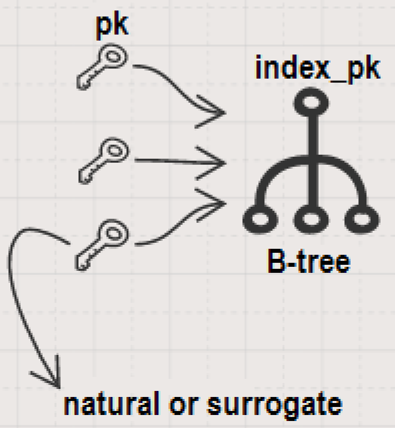
- **There are some fundamental differences between primary key and unique key - among others, a primary key cannot be NULL, while a unique key can be NULL.**

Primary Key

1 FK as well

Typically, PKs has a default index

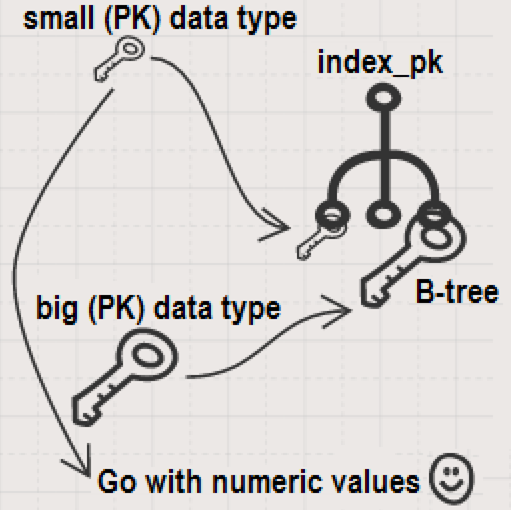
We can create other indexes as well



2 bytes matters!

Small PK result in a small index
Big PK result in a big index

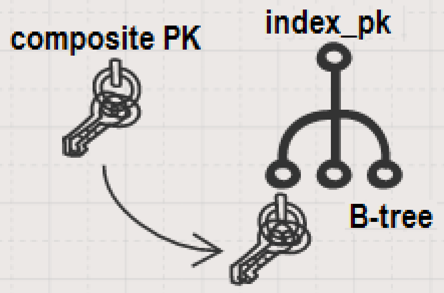
Keeping PK data type small is a good idea!



3 Are slow! Think to joins!

Composite PK results in big indexes

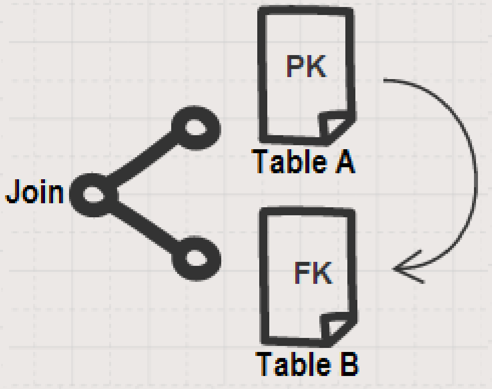
Try to avoid composite PK!



4 FK are usually indexed!

PK may be used for joins

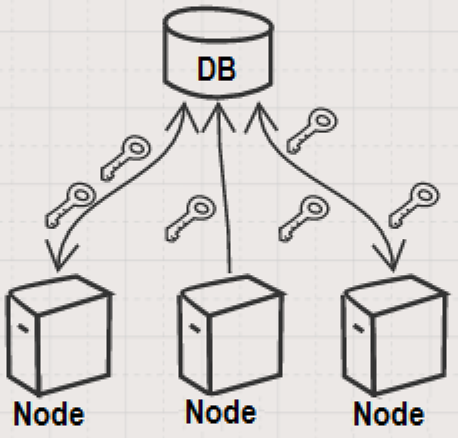
Just another reason to keep PK simple and small



5 Enforce uniqueness!

PK should be small, but still unique

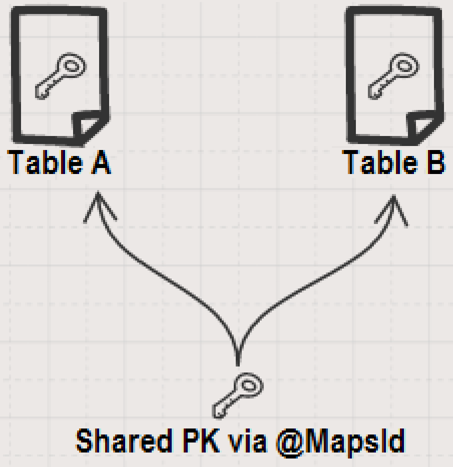
PK must be unique in high concurrent envs

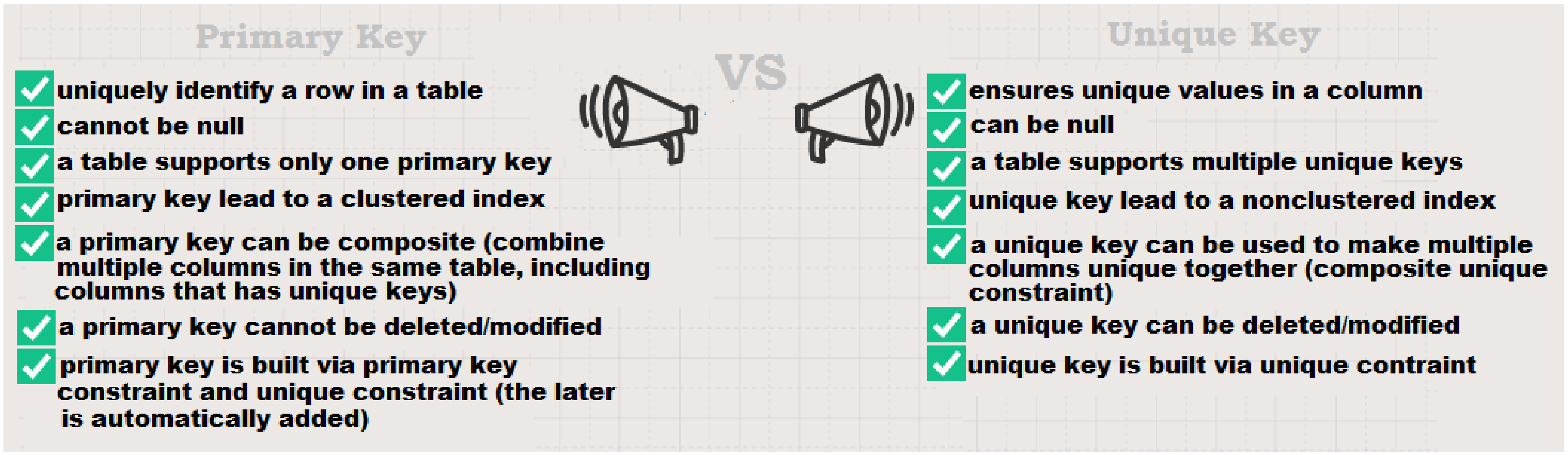


6 If it is possible, do it!

Share PK between tables

Sharing PK between tables reduce memory footprint by less indexes and no FK columns



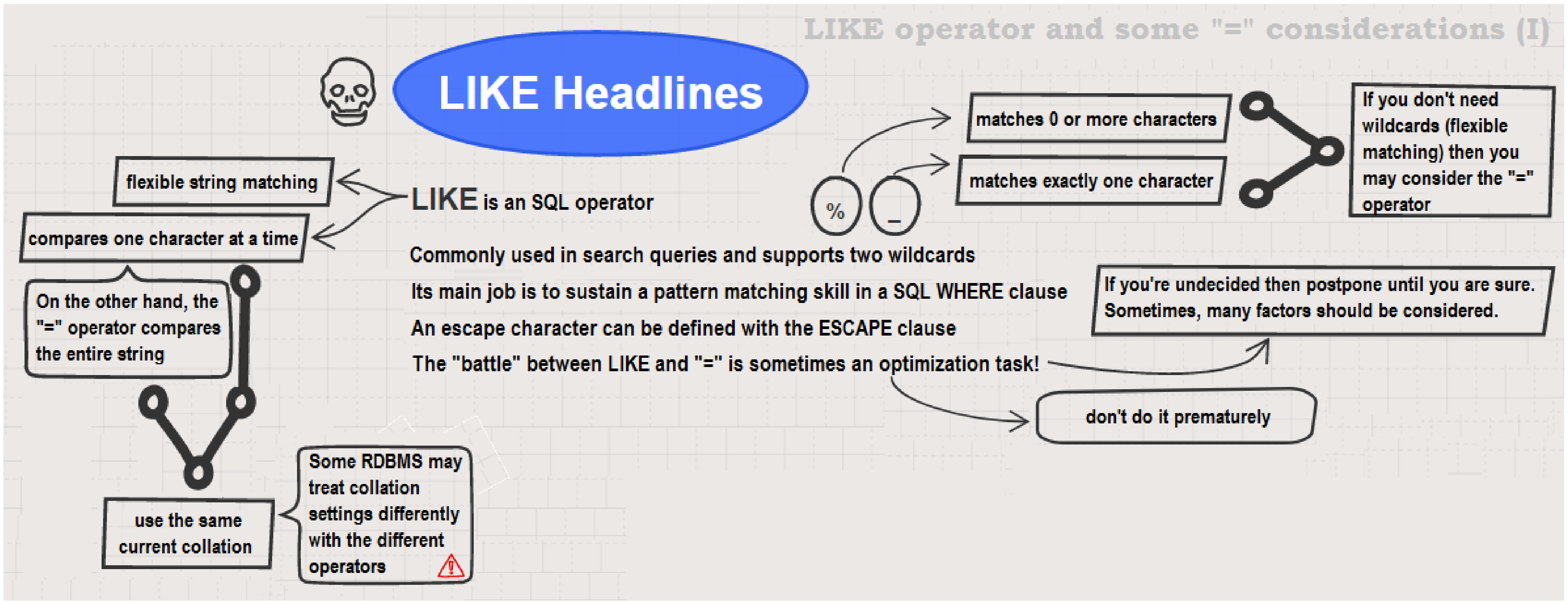


Primary Key	VS	Unique Key
✓ uniquely identify a row in a table		✓ ensures unique values in a column
✓ cannot be null		✓ can be null
✓ a table supports only one primary key		✓ a table supports multiple unique keys
✓ primary key lead to a clustered index		✓ unique key lead to a nonclustered index
✓ a primary key can be composite (combine multiple columns in the same table, including columns that has unique keys)		✓ a unique key can be used to make multiple columns unique together (composite unique constraint)
✓ a primary key cannot be deleted/modified		✓ a unique key can be deleted/modified
✓ primary key is built via primary key constraint and unique constraint (the later is automatically added)		✓ unique key is built via unique contraint

LIKE vs. equal (=)

The following two illustrations depicts `LIKE` headlines and `LIKE` versus equal (=) topic:

- **There is a debate about using `LIKE` operator versus equal (=).**
- **Mainly, the `LIKE` operator is useful for flexible string matching, while equal (=) is useful when we don't need wildcards.**
- **As a rule of thumb, in case of the `LIKE` operator, put wildcards as late as possible in the pattern matching and try to avoid wildcards on the first position in the pattern matching. Otherwise, you risk a full table/index scan and, as a consequence, a performance penalty.**



LIKE operator and some "=" considerations (I)

LIKE Headlines

LIKE is an SQL operator

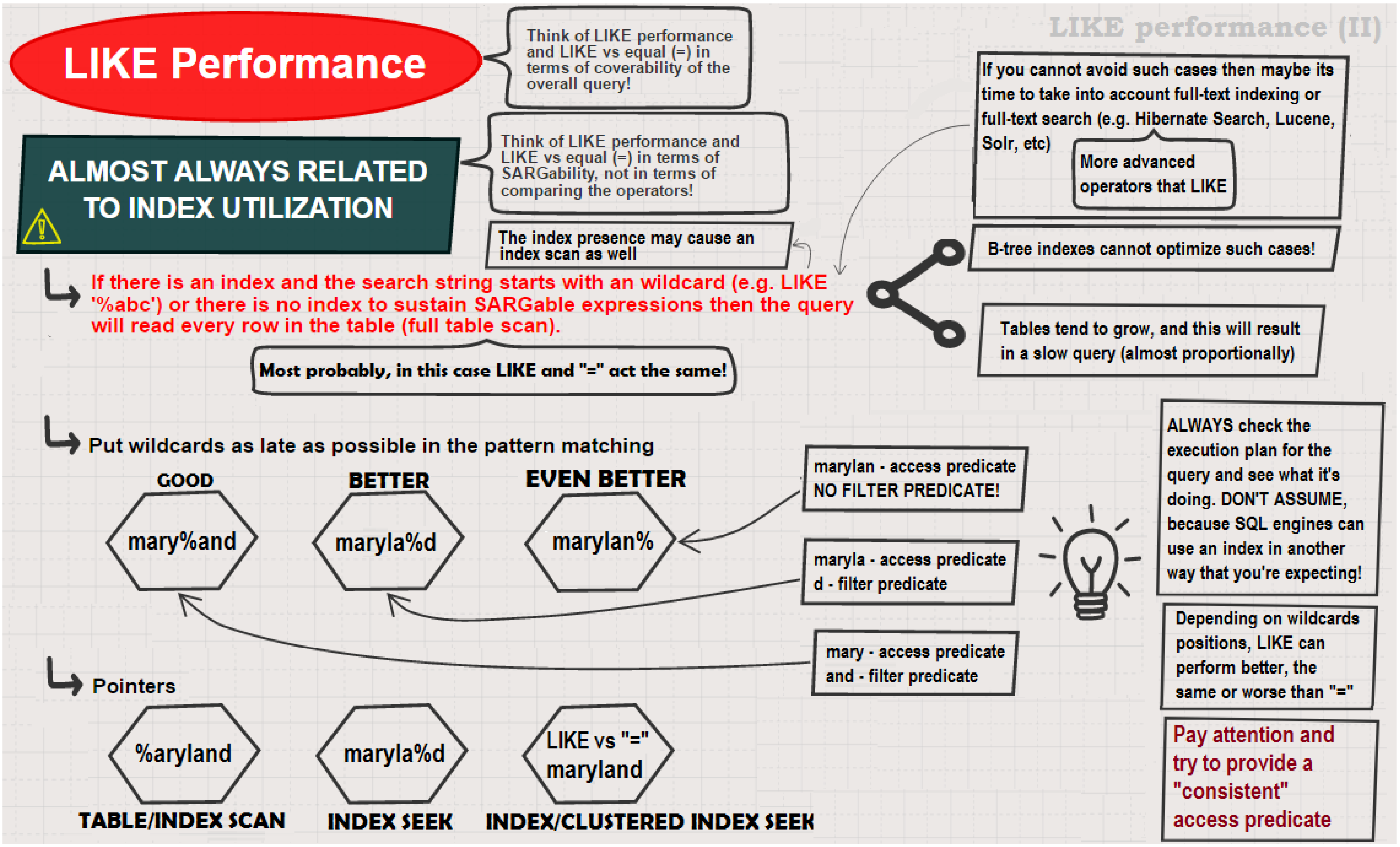
Commonly used in search queries and supports two wildcards

Its main job is to sustain a pattern matching skill in a SQL WHERE clause

An escape character can be defined with the ESCAPE clause

The "battle" between LIKE and "=" is sometimes an optimization task!

flexible string matching
compares one character at a time
On the other hand, the "=" operator compares the entire string
use the same current collation
Some RDBMS may treat collation settings differently with the different operators
%
_
matches 0 or more characters
matches exactly one character
If you don't need wildcards (flexible matching) then you may consider the "=" operator
If you're undecided then postpone until you are sure. Sometimes, many factors should be considered.
don't do it prematurely



LIKE performance (II)

LIKE Performance

Think of LIKE performance and LIKE vs equal (=) in terms of coverability of the overall query!

ALMOST ALWAYS RELATED TO INDEX UTILIZATION

Think of LIKE performance and LIKE vs equal (=) in terms of SARGability, not in terms of comparing the operators!

The index presence may cause an index scan as well

↳ If there is an index and the search string starts with an wildcard (e.g. LIKE '%abc') or there is no index to sustain SARGable expressions then the query will read every row in the table (full table scan).

Most probably, in this case LIKE and "=" act the same!

If you cannot avoid such cases then maybe its time to take into account full-text indexing or full-text search (e.g. Hibernate Search, Lucene, Solr, etc)

More advanced operators that LIKE

B-tree indexes cannot optimize such cases!

Tables tend to grow, and this will result in a slow query (almost proportionally)

↳ Put wildcards as late as possible in the pattern matching

GOOD	BETTER	EVEN BETTER
mary%and	maryla%d	marylan%

marylan - access predicate
NO FILTER PREDICATE!

maryla - access predicate
d - filter predicate

mary - access predicate
and - filter predicate

ALWAYS check the execution plan for the query and see what it's doing. DON'T ASSUME, because SQL engines can use an index in another way that you're expecting!

Depending on wildcards positions, LIKE can perform better, the same or worse than "="

Pay attention and try to provide a "consistent" access predicate

↳ Pointers

%aryland	maryla%d	LIKE vs "=" maryland
TABLE/INDEX SCAN	INDEX SEEK	INDEX/CLUSTERED INDEX SEEK

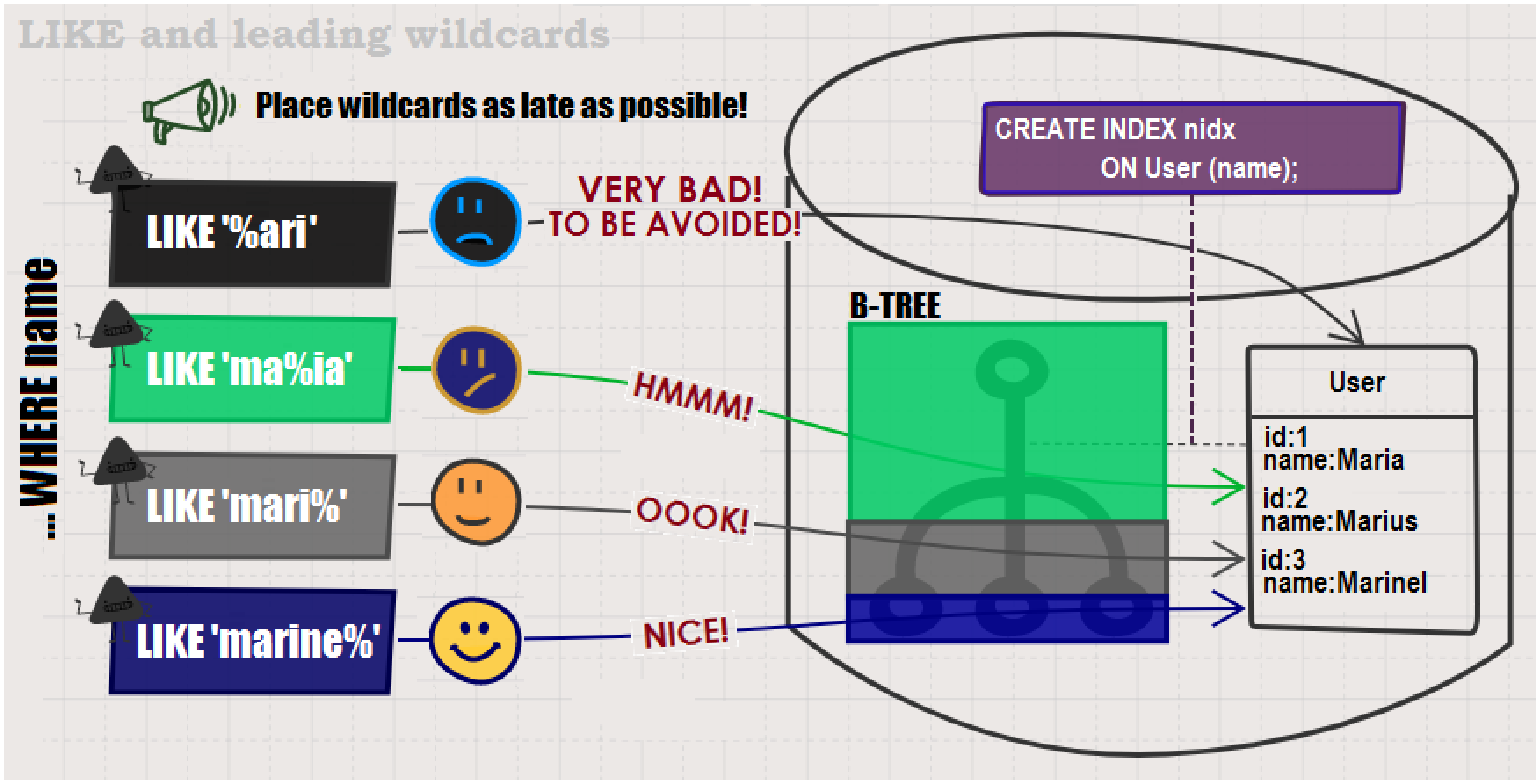
LIKE wildcards usage

The following illustration depicts the `LIKE` wildcards usage:

- **The positions of wildcards in the `LIKE` expressions can have a significant impact on choosing the Execution Plan and on index range that need to be scanned.**

- **As a rule of thumb, place wildcards as later as possible and never on the first position. Since only the part before the first wildcard can be used as an access predicate, using the wildcard on the first position will lead to an entire table scan (no access predicate).**

- **Scanning the entire table or a big index range can be a performance penalty.**

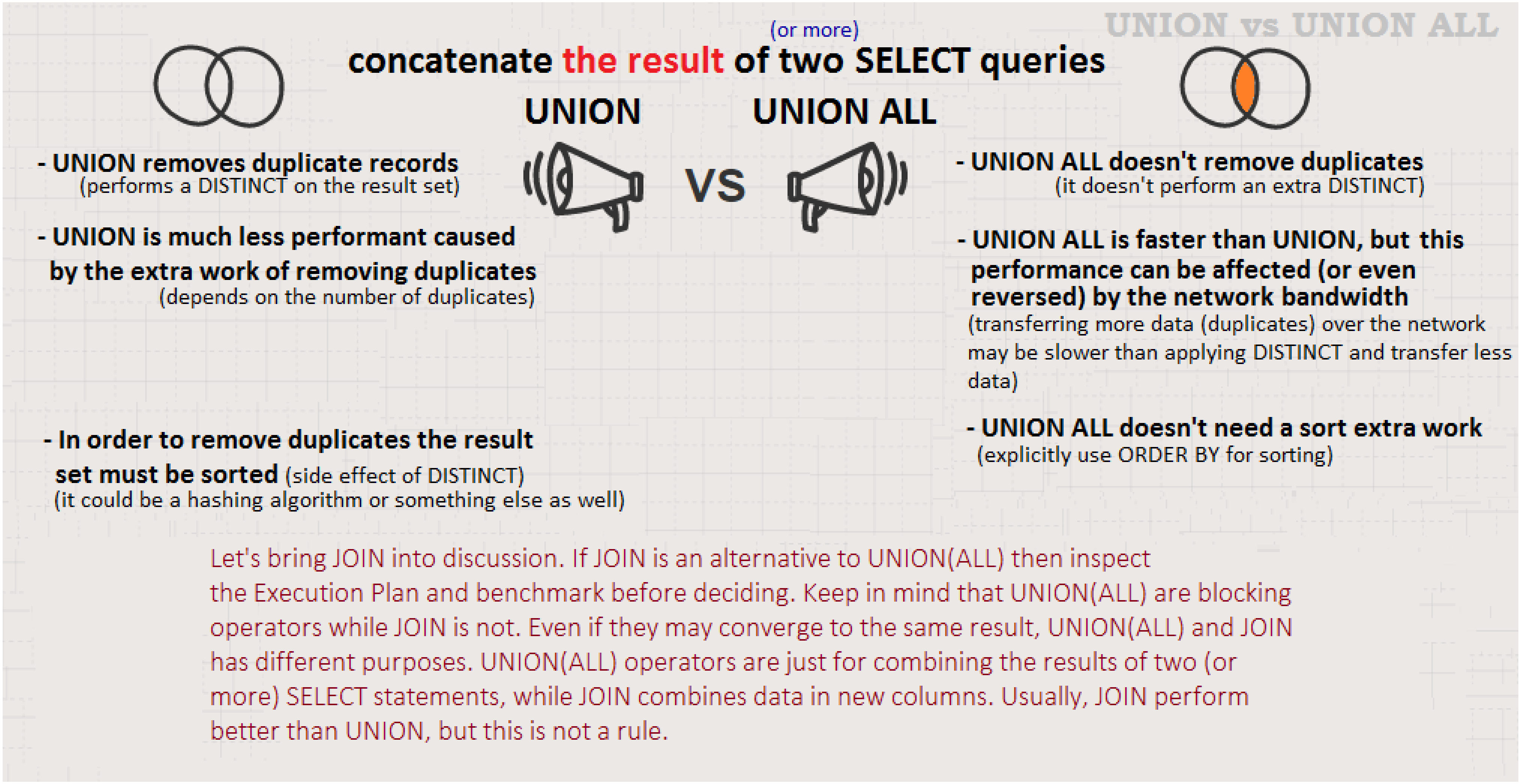

LIKE and leading wildcards
Place wildcards as late as possible!
... WHERE name
LIKE '%ari'
VERY BAD!
TO BE AVOIDED!
LIKE 'ma%ia'
HMMM!
LIKE 'mari%'
OOOK!
LIKE 'marine%'
NICE!
CREATE INDEX nidx
ON User (name);
B-TREE
User
id:1
name:Maria
id:2
name:Marius
id:3
name:Marinel

UNION vs. UNION ALL and JOIN flavors

The following illustrations depicts the `UNION` versus `UNION ALL` and `JOIN` topic:

- **Concatenating the result of two (or more) `SELECT` queries can be accomplished in different ways.**

- **Follow the below headlines to ensure that you make the best choice from the performance perspective.**



UNION vs UNION ALL

(or more)

concatenate the result of two SELECT queries

UNION VS **UNION ALL**

- **UNION removes duplicate records**
  (performs a DISTINCT on the result set)

- **UNION is much less performant caused by the extra work of removing duplicates**
  (depends on the number of duplicates)

- **In order to remove duplicates the result set must be sorted** (side effect of DISTINCT)
  (it could be a hashing algorithm or something else as well)

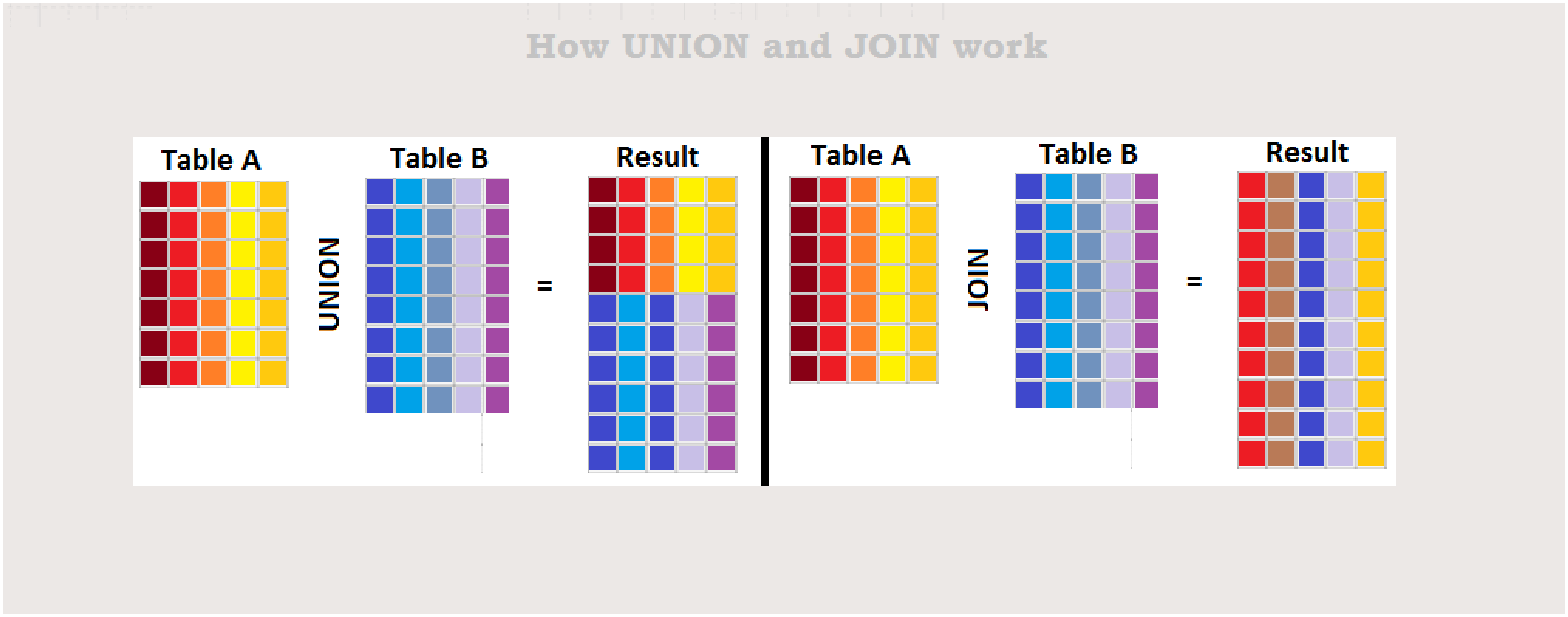
- **UNION ALL doesn't remove duplicates**
  (it doesn't perform an extra DISTINCT)

- **UNION ALL is faster than UNION, but this performance can be affected (or even reversed) by the network bandwidth**
  (transferring more data (duplicates) over the network may be slower than applying DISTINCT and transfer less data)

- **UNION ALL doesn't need a sort extra work**
  (explicitly use ORDER BY for sorting)

Let's bring JOIN into discussion. If JOIN is an alternative to UNION(ALL) then inspect the Execution Plan and benchmark before deciding. Keep in mind that UNION(ALL) are blocking operators while JOIN is not. Even if they may converge to the same result, UNION(ALL) and JOIN has different purposes. UNION(ALL) operators are just for combining the results of two (or more) SELECT statements, while JOIN combines data in new columns. Usually, JOIN perform better than UNION, but this is not a rule.

A famous and very inspired representation of how `JOIN` and `UNION` work is as in the figures below:



Part II

RDBMS Headlines

Result Set

Limit the result set

Result set scrollability

Limit the result set

The following illustration depicts how to limit the result set:

- **We need to control the size of the result set and always be aware about the size evolution in time (especially increases).**

- **We can control the size of the result set in two ways:**
  - **SQL restriction**
  - **JDBC max rows**

- **SQL restriction is recommended against JDBC max rows since it will help the database server to choose the best Execution Plan, and therefore, the best performance trend.**

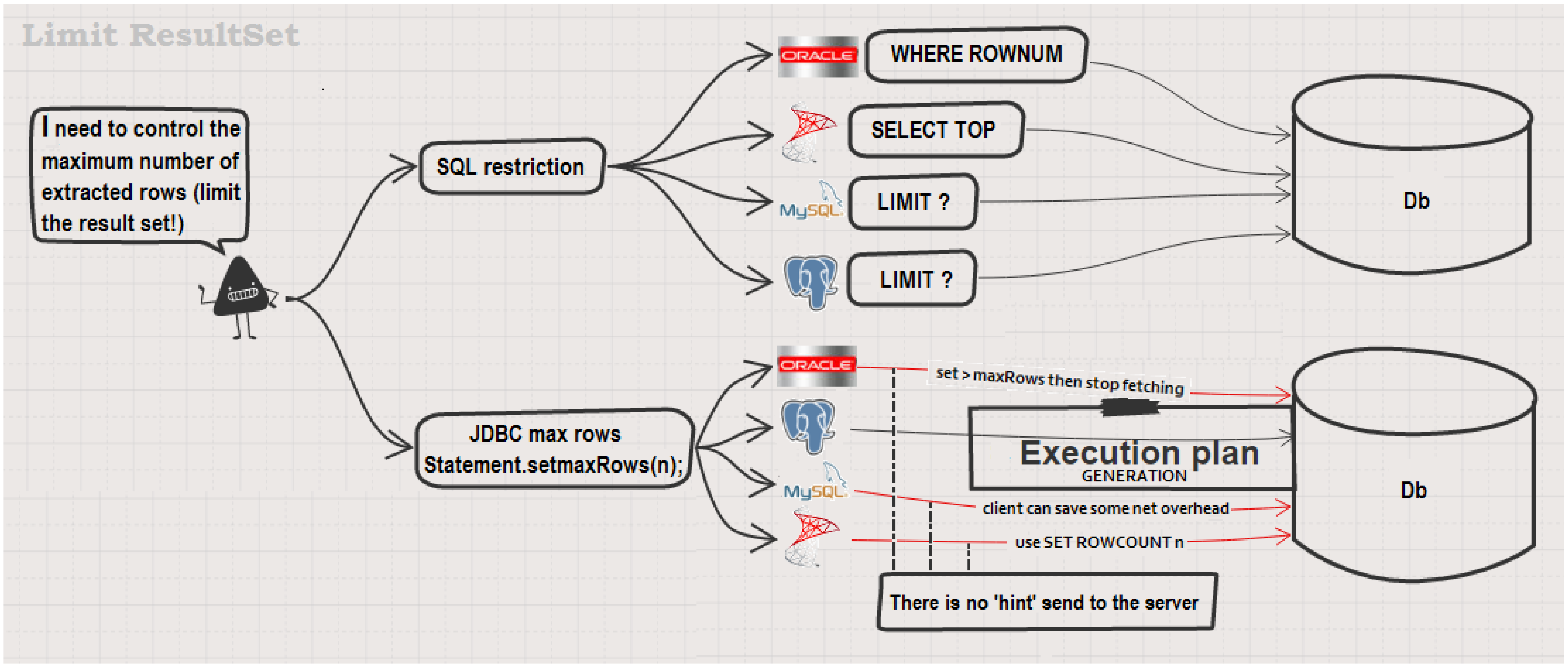
- **JDBC max rows affects the Execution Plan decision in PostgreSQL, but it won't trigger any kind of "hint" in case of the Oracle, SQL Server and MySQL servers.**

- **As a rule of thumb, always control the result set size and keep in mind that this will "grow" with respect to the underlying table data.**

- **Without controlling the size of the result set, you may end up with long transactions that will cause a dramatic performance degradation.**

Applications:

- **How To Use Query Creation Mechanism For JPA To Limit Result Size**


Limit ResultSet
I need to control the maximum number of extracted rows (limit the result set!)
SQL restriction
WHERE ROWNUM
SELECT TOP
LIMIT ?
LIMIT ?
Db
JDBC max rows
Statement.setmaxRows(n);
set > maxRows then stop fetching
Execution plan
GENERATION
client can save some net overhead
use SET ROWCOUNT n
There is no 'hint' send to the server
Db

Result set scrollability

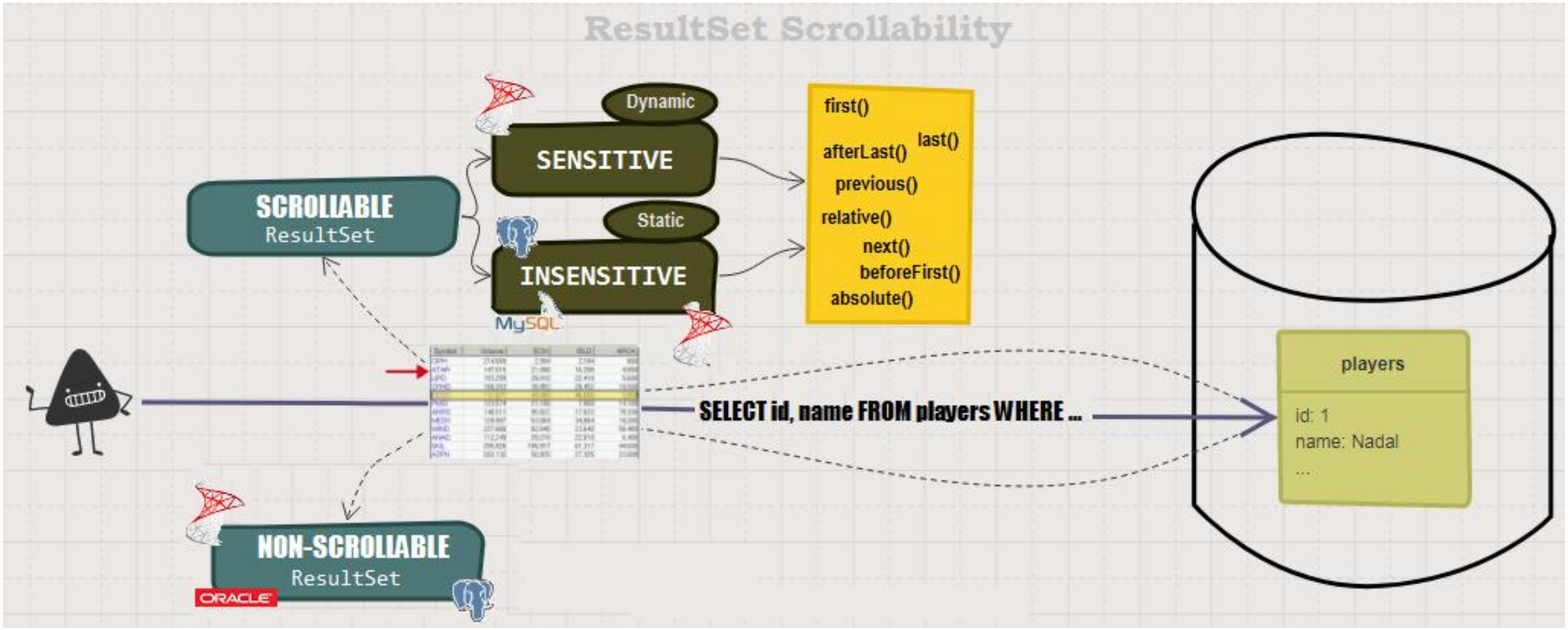
The following illustration depicts result set scrollability:

- **The result set can be traversed by using an application-level cursor.**

- **Result set can be *scrollable* or *non-scrollable.***

- **For *non-scrollable* result set we have the following headlines:**
  - **the cursor move only in forward direction (*forward-only* application level cursor);**
  - **the cursor cannot be moved in a random approach;**
  - **If the result set is big then this approach may lead to performance penalties since if we want to move to the *n*th record then we need to perform *n+1* iteration;**
  - **PostgreSQL, Oracle and SQL Server support *non-scrollable* result set;**

- **For *scrollable* result set we have the following headlines:**
  - **the cursor can be moved in both directions (forward and backward) by using the following methods (`first()`, `last()`, `next()`, `previous()`, `absolute()`, `relative()`, `afterLast()` and `beforeFirst()`);**
  - **the cursor can be moved in a random approach;**
  - **since we can move directly to any record we don't have performance penalties;**
  - ***scrollable* can be *sensitive* (the result set is fetched dynamically and it can reflect concurrent changes (this is known as a *dynamic view*)) or *insensitive* (the data is fetched entirely before being iterated (this is known as a *static view* of the current result set));**
  - ***sensitive* cursor is supported by SQL Server;**
  - ***insensitive* cursor is supported by SQL Server, PostgreSQL and MySQL;**


ResultSet Scrollability
Dynamic
SENSITIVE
Static
INSENSITIVE
SCROLLABLE
ResultSet
first()
last()
afterLast()
previous()
relative()
next()
beforeFirst()
absolute()
MySQL
SELECT id, name FROM players WHERE ...
players
id: 1
name: Nadal
...
NON-SCROLLABLE
ResultSet
ORACLE

Transactions

Atomicity (ACID)

Consistency(ACID)

Isolation (ACID)

Durability(ACID)

Transaction rollback

Deadlock

Non-repeatable reads vs. repeatable reads

Two-phase locking (2PL)

Dirty read

Dirty write

Lost update

Phantom read

Read skew

Write skew

Read Uncommitted

Read Committed

Repeatable Reads

Serializable

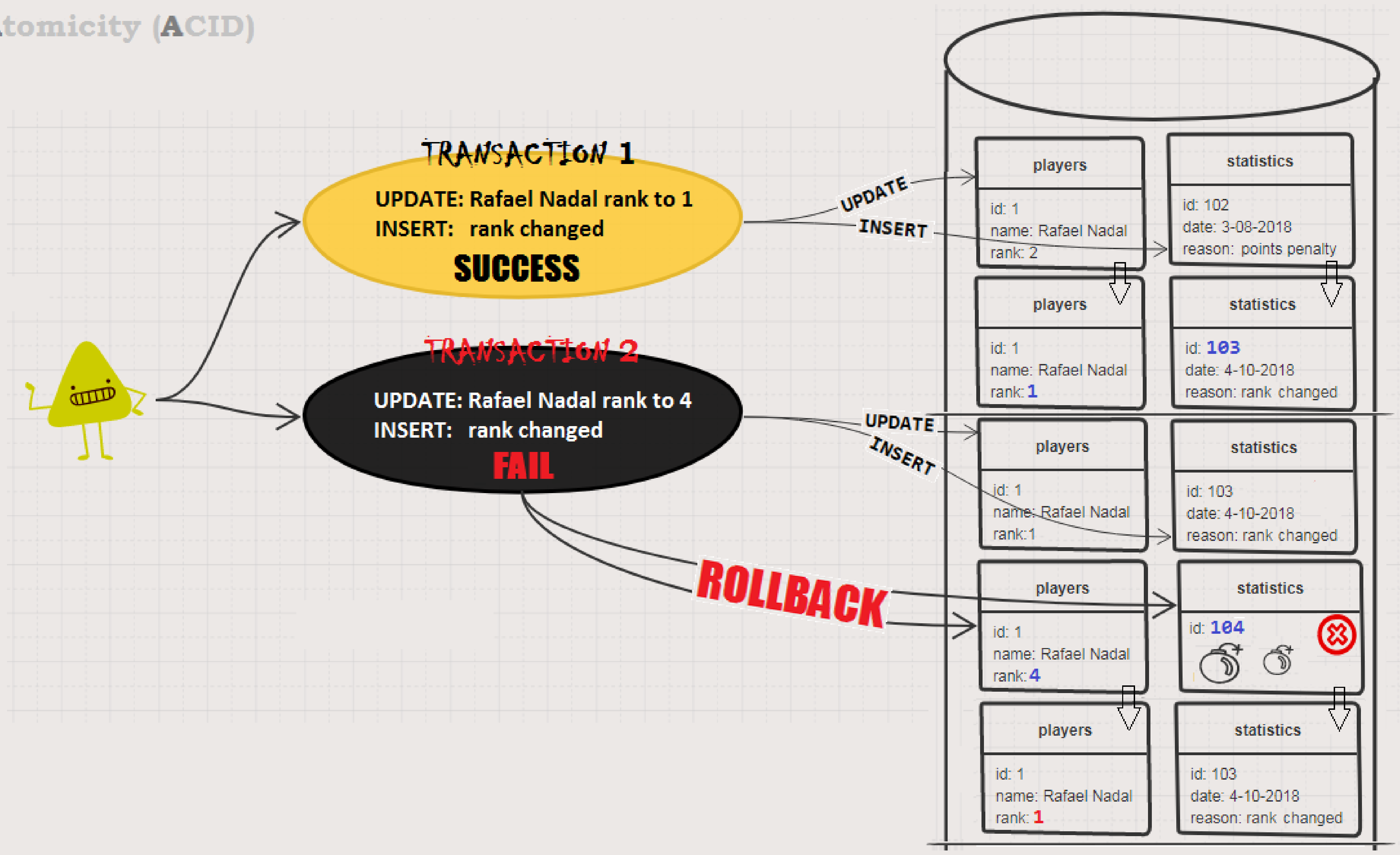
Atomicity (ACID)

The following illustration depicts Atomicity from ACID:

- **By *unit-of-work* we understand a bunch of individual operations grouped together (typically, these operations are logically related).**

- **Atomicity is the property that guarantees that a *unit-of-work* is executed with success or not against the database. In other words, if a single operation of an *unit-of-work* cannot be executed successfully, then the action is aborted and the already performed operations (every executed statement) must be rollback automatically.**

Atomicity (ACID)



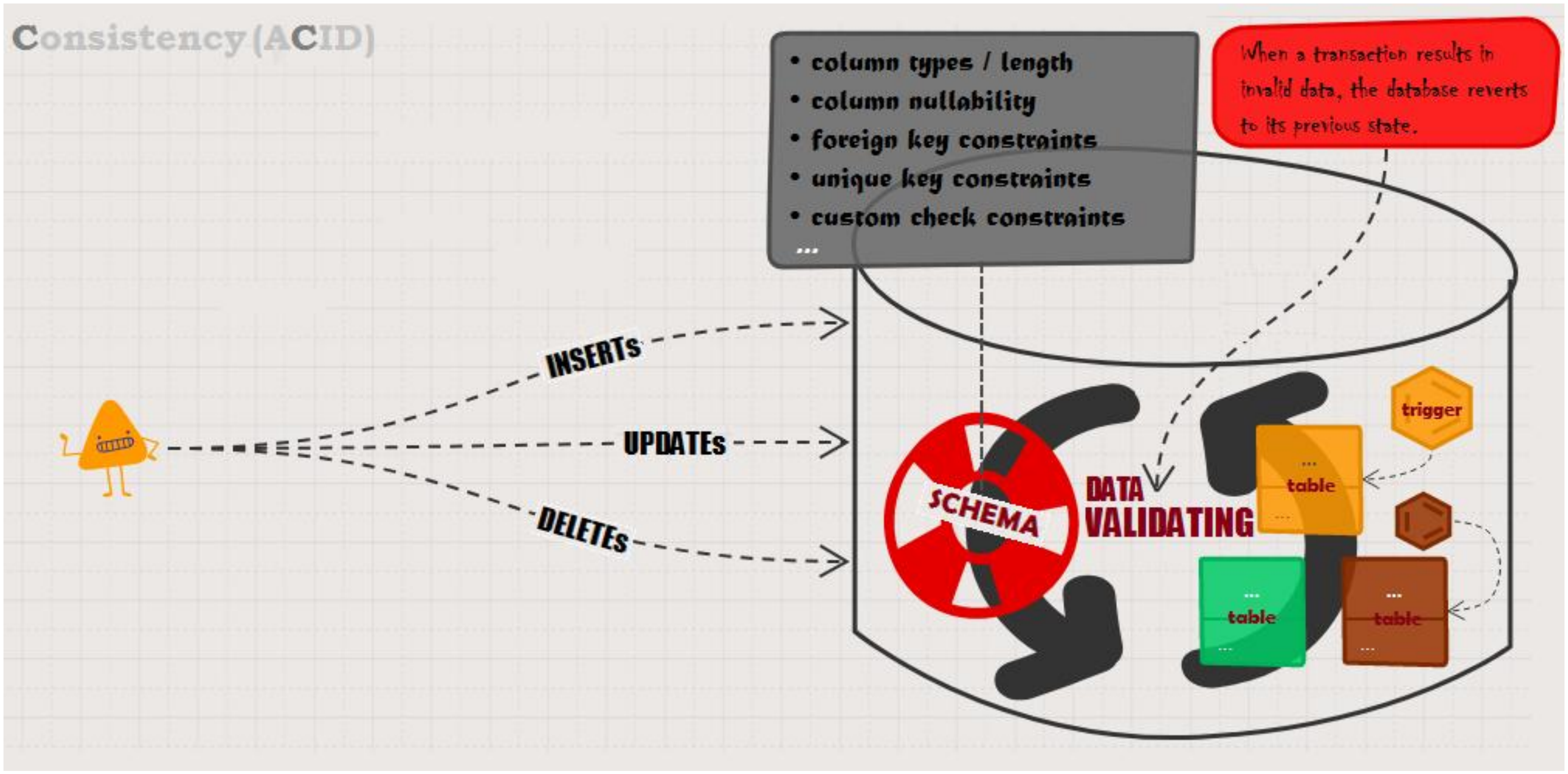
Consistency (ACID)

The following illustration depicts Consistency from ACID:

- **Consistency is the property that ensures that only valid data is written to the database. In other words, any transaction will bring the database from one valid state to another valid state.**

- **Valid data is the data that follows all the defined rules and constrains.**

- **When a transaction results in invalid data, the database reverts to its previous state, which abides by all customary rules and constraints.**



Consistency (ACID)

- column types / length
- column nullability
- foreign key constraints
- unique key constraints
- custom check constraints
- ...

When a transaction results in invalid data, the database reverts to its previous state.

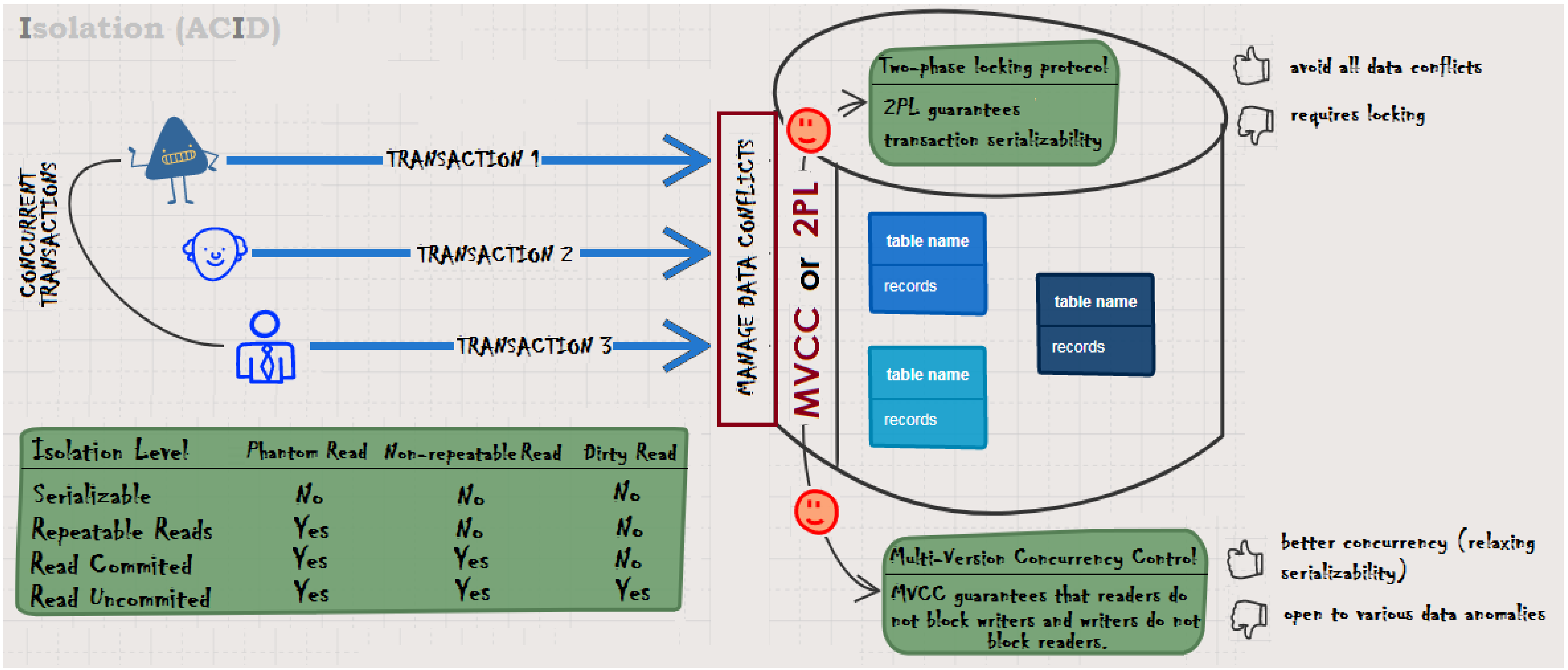
Isolation (ACID)

The following illustration depicts Isolation from ACID:

- **Isolation is the property responsible to manage data conflicts (data collisions) that may occur during concurrent transactions.**

- **Among the strategies develop to manage data conflicts, we highlight the following two:**
  - ***two-phase locking (2PL)*: this strategy requires locking in order to obtain serializable transactions, and typically has a poor performance in comparison with MVCC (think that locking introduces contention, which, in turn, limits concurrency and scalability);**
  - ***multi-version concurrency control (MVCC)*: this strategy is based on detecting conflicts and provide better concurrency. Most database vendors has adopted this strategy which is accepting different data anomalies (phenomena);**

- **SQL-92 version introduced multiple isolation levels:**
  - **Read Uncommitted**
  - **Read Committed**
  - **Repeatable Read**
  - **Serializable**


Isolation (ACID)
CONCURRENT TRANSACTIONS
TRANSACTION 1
TRANSACTION 2
TRANSACTION 3
MANAGE DATA CONFLICTS
MVCC or 2PL
Two-phase locking protocol
2PL guarantees transaction serializability
avoid all data conflicts
requires locking
table name
records
table name
records
table name
records
Multi-Version Concurrency Control
MVCC guarantees that readers do not block writers and writers do not block readers.
better concurrency (relaxing serializability)
open to various data anomalies

Isolation Level	Phantom Read	Non-repeatable Read	Dirty Read
Serializable	No	No	No
Repeatable Reads	Yes	No	No
Read Commited	Yes	Yes	No
Read Uncommited	Yes	Yes	Yes

Durability (ACID)

The following illustration depicts Durability from ACID:

- **Durability is the property that guarantees that the changes performed by all committed transactions are permanent.**

- **Durability is directly related to recoverability of the database in case of system failures, restarts, etc.**

- **Durability is achieved via *logs*. Most databases support *undo* and *redo log.* Recoverability needs committed changes only.**

- **Typically, the *undo log* is used for providing transaction rollback support and is not involved in recoverability process.**

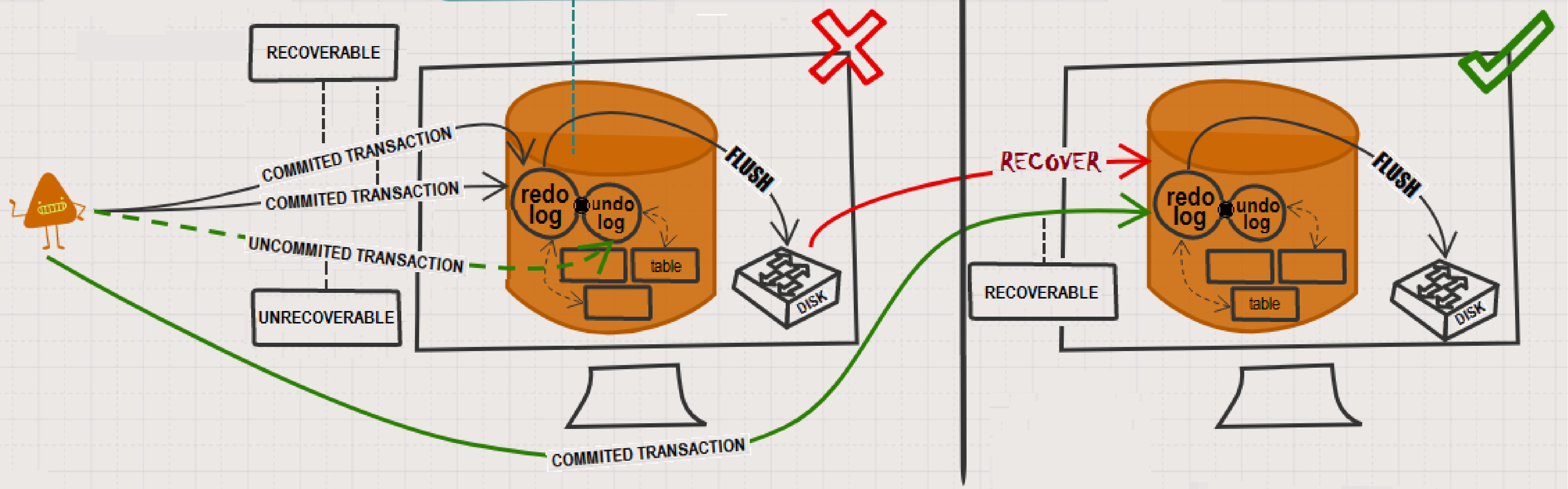
- **The *redo log* is used to persist all current changes. Typically, this action take place when a transaction in committed.**

- **How *undo/redo* logs are implemented, how they are used internally, when they are flushed to disk, and so on and forth is vendor specific.**

- **The picture below is a blueprint that may be slightly different depending on database vendor.**

Durability (ACID) (blueprint)

Notice that how redo/undo logs are flushed and used for recovery is vendor specific.



Oracle	SQL Server	PostgreSQL	MySQL
uses at least two redo files, but only one is active for collecting the log buffer entries	combines the redo and undo logs in a data structure called transaction log	use a Write-Ahead Log	uses mini transaction buffer, and a global redo buffer which is flushed to disk during commit

Transaction rollback

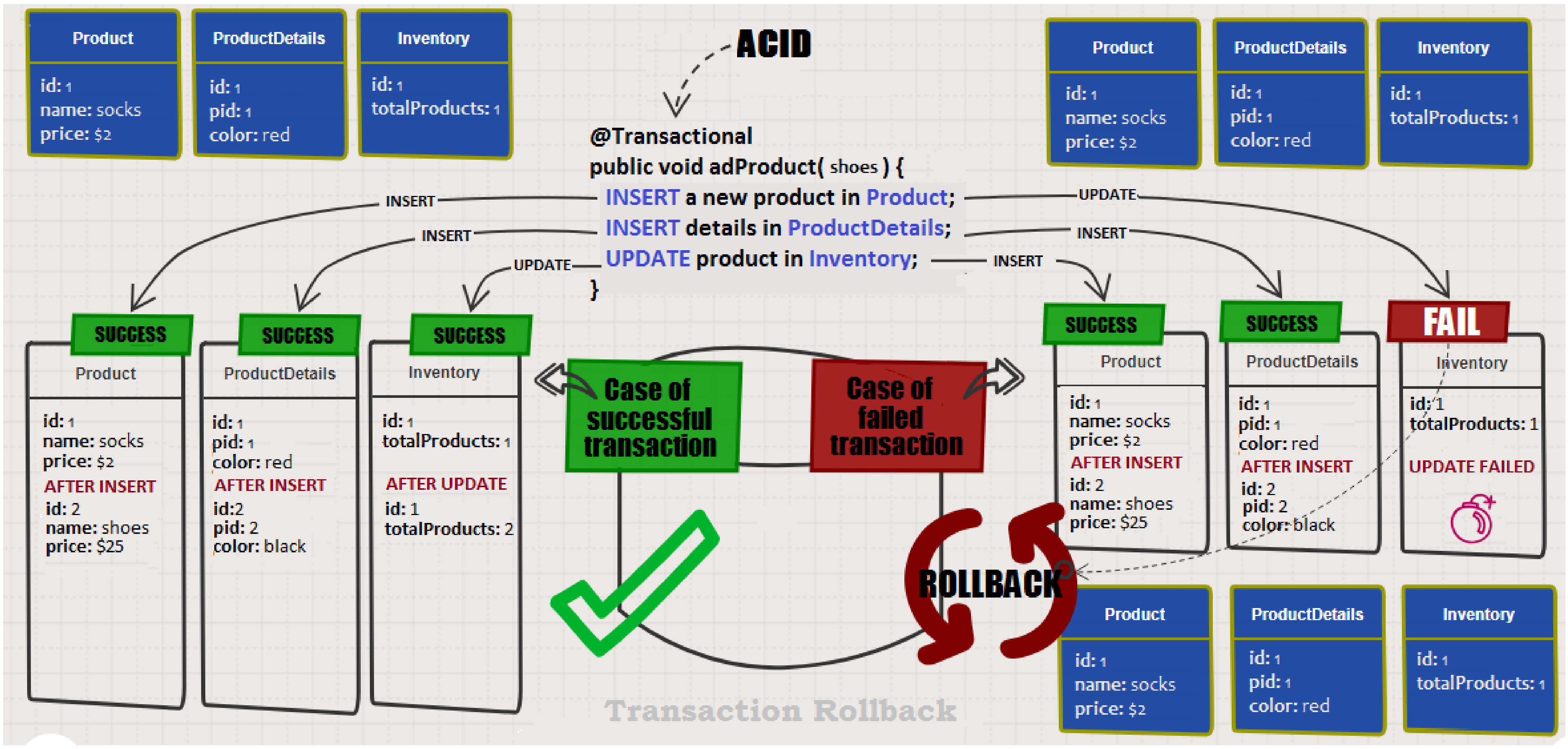
The following illustration depicts transaction rollback:

- **First of all, use explicit transactions even if you fetch *read-only* data.**

- ***Rollback* is a technique used to ensure that database never contains the result of partial operations. If one of the operations that is part of a transaction fails, the *rollback* mechanism occurs to restore the database to its original state (this is the state before the transaction was started). If no error occur during the transaction lifespan, the entire set of operations (statements) is committed to the database.**

- ***Rollback* should be explicitly called when we manage the transaction boundaries (commonly, in such cases, we also start, rollback or commit the transaction via some explicit dedicated methods).**

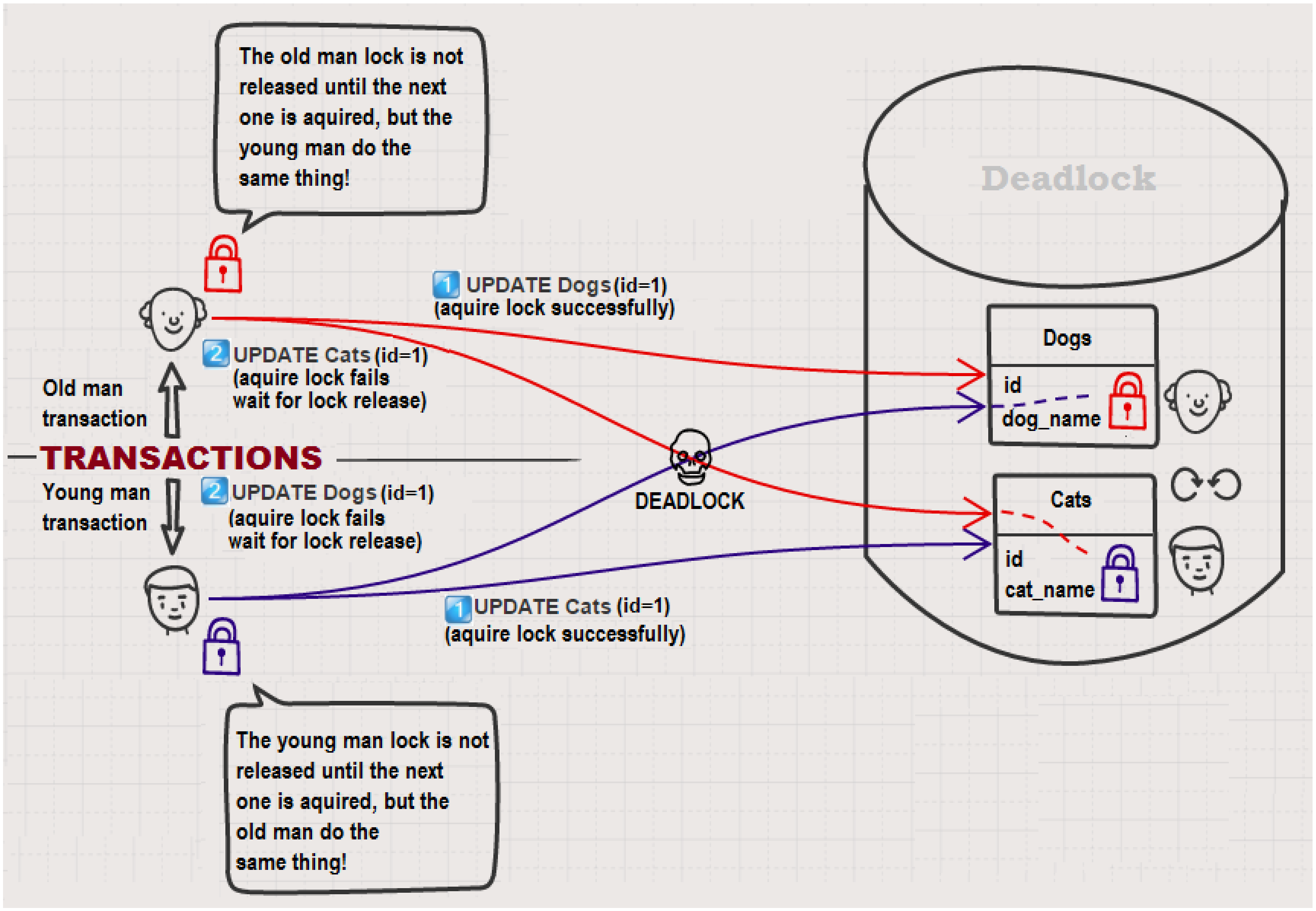
- ***Rollback* can be implicitly/automatically called as well (e.g., like in the case of `@Transactional` from Spring, which will rollback changes by default in case of any runtime exception).**


Product
id: 1
name: socks
price: $2
ProductDetails
id: 1
pid: 1
color: red
Inventory
id: 1
totalProducts: 1
ACID
@Transactional
public void adProduct(shoes) {
INSERT a new product in Product;
INSERT details in ProductDetails;
UPDATE product in Inventory;
}
INSERT
INSERT
UPDATE
UPDATE
INSERT
INSERT
SUCCESS
Product
id: 1
name: socks
price: $2
AFTER INSERT
id: 2
name: shoes
price: $25
SUCCESS
ProductDetails
id: 1
pid: 1
color: red
AFTER INSERT
id:2
pid: 2
color: black
SUCCESS
Inventory
id: 1
totalProducts: 1
AFTER UPDATE
id: 1
totalProducts: 2
Case of successful transaction
Case of failed transaction
ROLLBACK
FAIL
UPDATE FAILED
Transaction Rollback

Deadlock

The following illustration depicts the deadlock topic:

- **In a deadlock, two transactions are blocking each other.**
- **Transaction A holds a lock to resource A, but it doesn't release it until it will acquire a lock to resource B that is currently locked by transaction B.**
- **In the same time, transaction B holds a lock to resource B, but it doesn't release it until it will acquire a lock to resource A that is currently locked by transaction A.**
- **The application (data access layer) is responsible to manage the locks acquisition/release in order to avoid deadlocks, and implicitly performance issues.**
- **Without application support, you need to rely on the database capability of detecting and solving deadlocks (commonly, by aborting one of the transactions involved in a deadlock).**

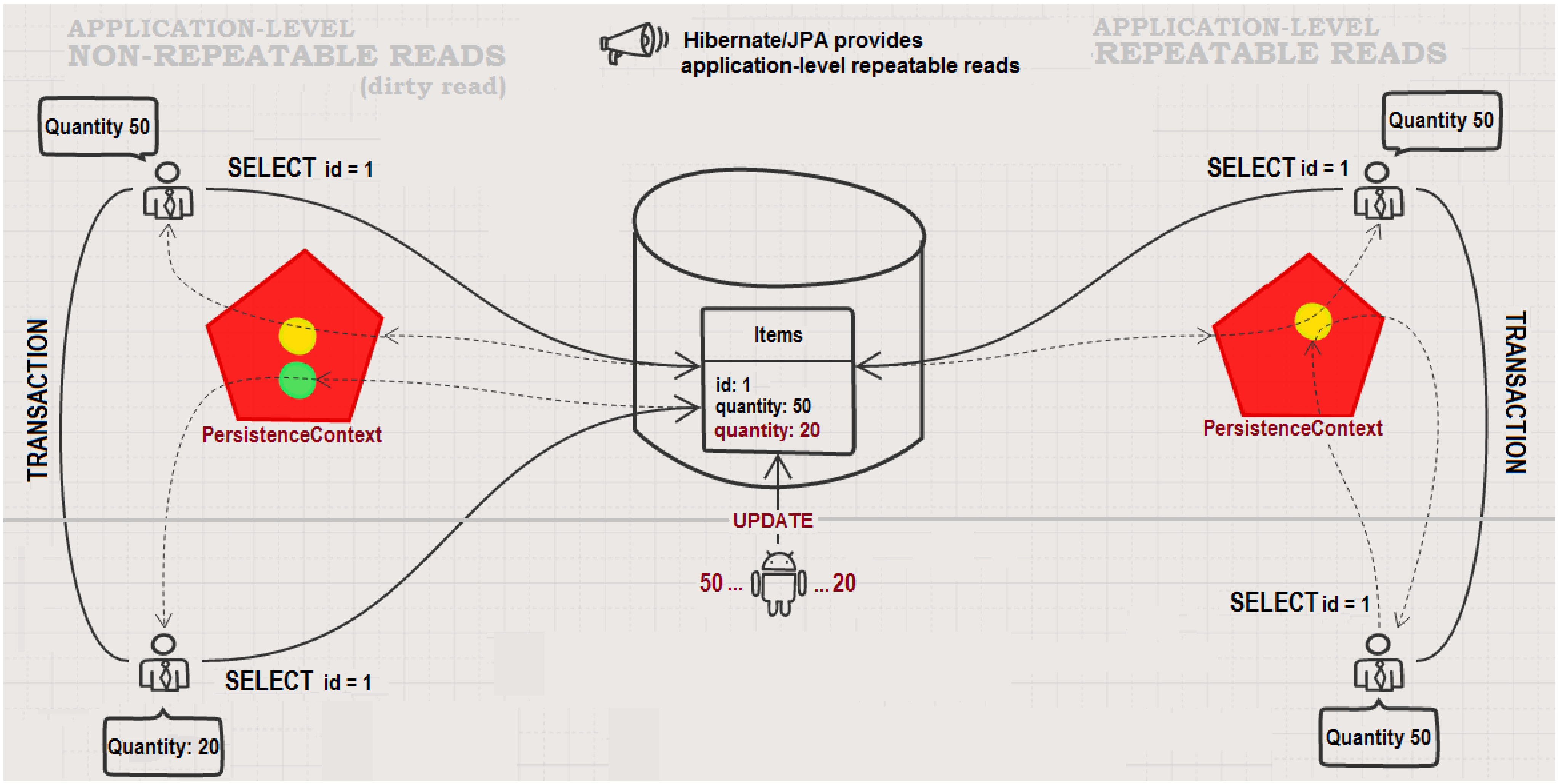

The old man lock is not released until the next one is aquired, but the young man do the same thing!
1 UPDATE Dogs(id=1) (aquire lock successfully)
2 UPDATE Cats (id=1) (aquire lock fails wait for lock release)
Old man transaction
TRANSACTIONS
Young man transaction
2 UPDATE Dogs (id=1) (aquire lock fails wait for lock release)
DEADLOCK
1 UPDATE Cats (id=1) (aquire lock successfully)
The young man lock is not released until the next one is aquired, but the old man do the same thing!
Deadlock
Dogs
id
dog_name
Cats
id
cat_name

Application-level non-repeatable reads vs. application-level repeatable reads

The following illustration depicts the application-level non-repeatable reads (dirty read) versus application-level repeatable reads topic:

- **Hibernate/JPA provides application-level repeatable reads via the First-Level Cache (Persistent Context).**

- **Repeatable Read isolation level is the traditional approach for preventing lost updates (doesn't work for conversations that spans over several requests).**


APPLICATION-LEVEL
NON-REPEATABLE READS
(dirty read)
Hibernate/JPA provides application-level repeatable reads
APPLICATION-LEVEL
REPEATABLE READS
Quantity 50
SELECT id = 1
TRANSACTION
PersistenceContext
Items
id: 1
quantity: 50
quantity: 20
UPDATE
50 ...
... 20
SELECT id = 1
Quantity: 20
Quantity 50
SELECT id = 1
TRANSACTION
PersistenceContext
SELECT id = 1
Quantity 50

Two-phase locking (2PL)

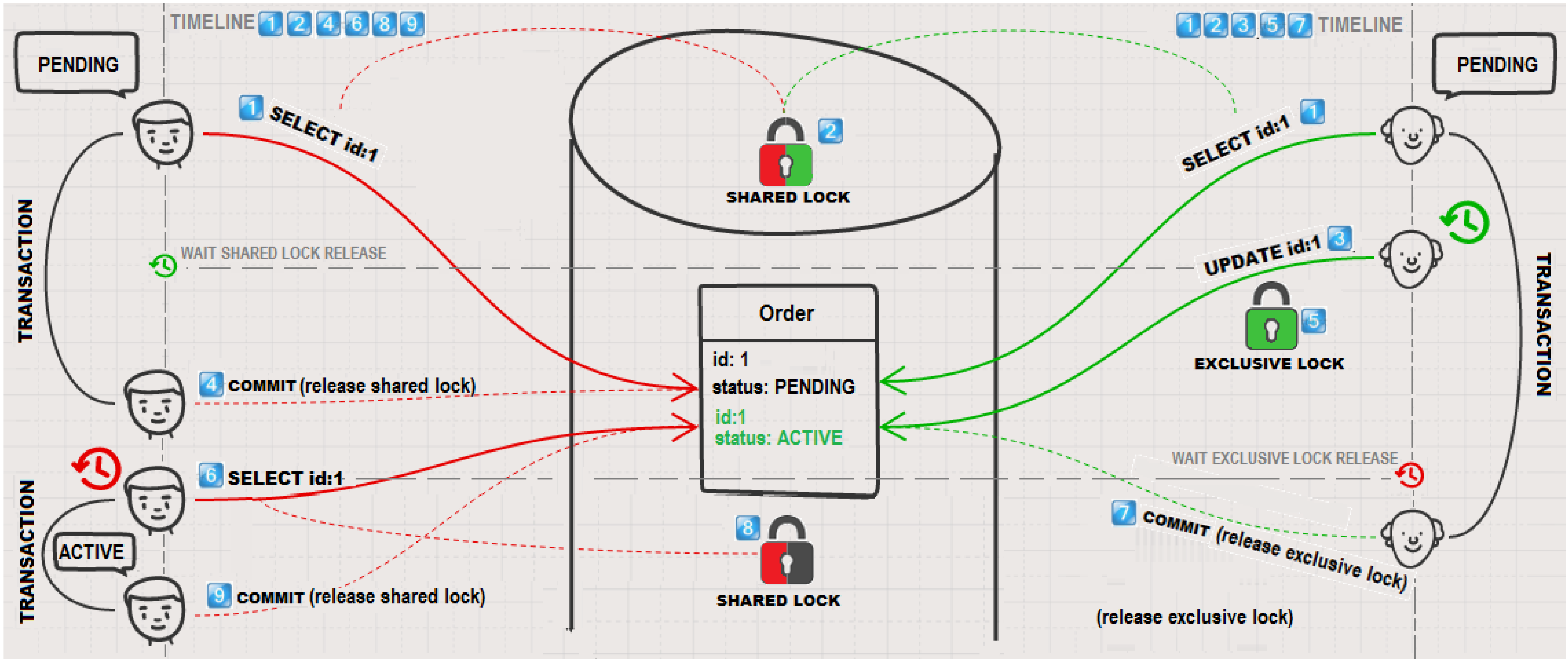
The following illustration depicts the 2PL topic:

- **The two-phase locking protocol (2PL) sustain serializable transactions via locking (shared and exclusive locks) and prevents conflicts.**

- **From performance penalty perspective, the lock contention can play a major role. Check the picture below and notice how transactions must wait for locks to be released.**

- **The solution (embraced by the majority of database vendors) consist in using Multi-Version Concurrency Control (MVCC) instead of 2PL.**

- **In MVCC, only writers can block other concurrent writers.**



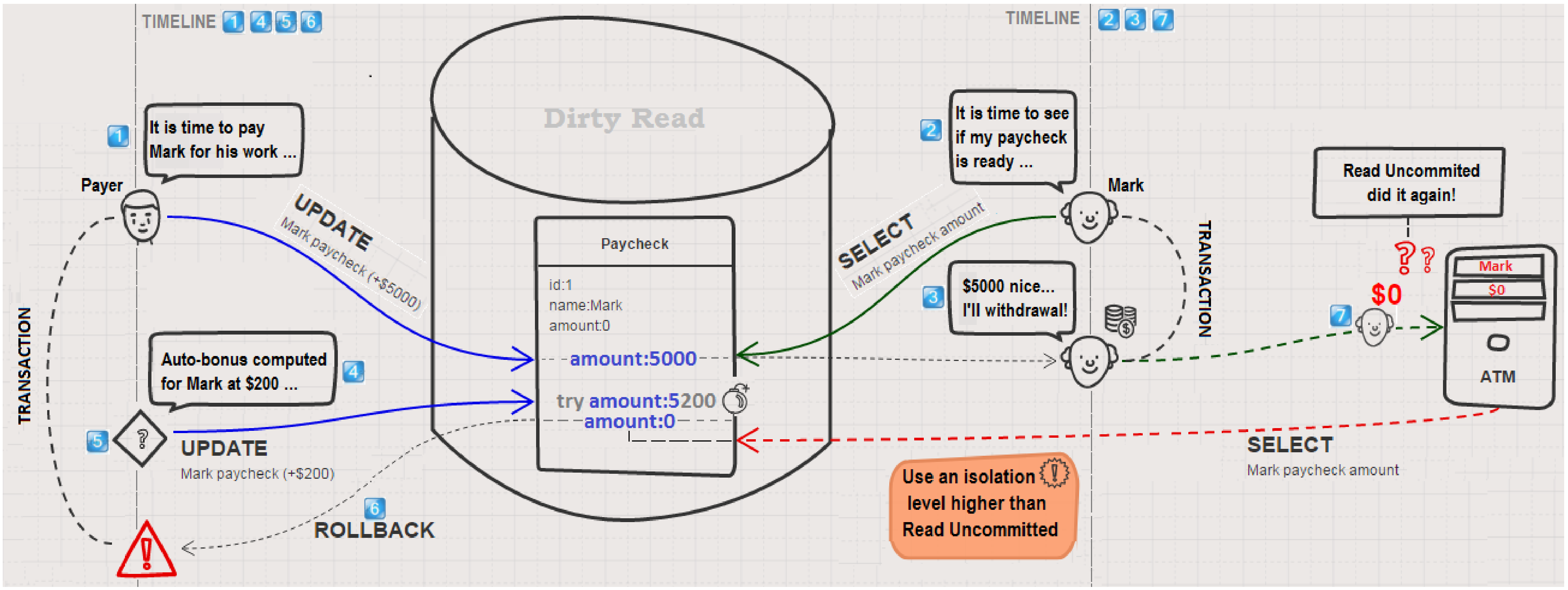
Shared locks applies to selects and allows concurrent reads, but no other transaction is allowed to update, delete, or get an exclusive lock.
Exclusive locks applies to update and delete and prevent concurrent reads and writes.

Two-phase locking

Dirty read

The following illustration depicts the dirty-read anomaly:

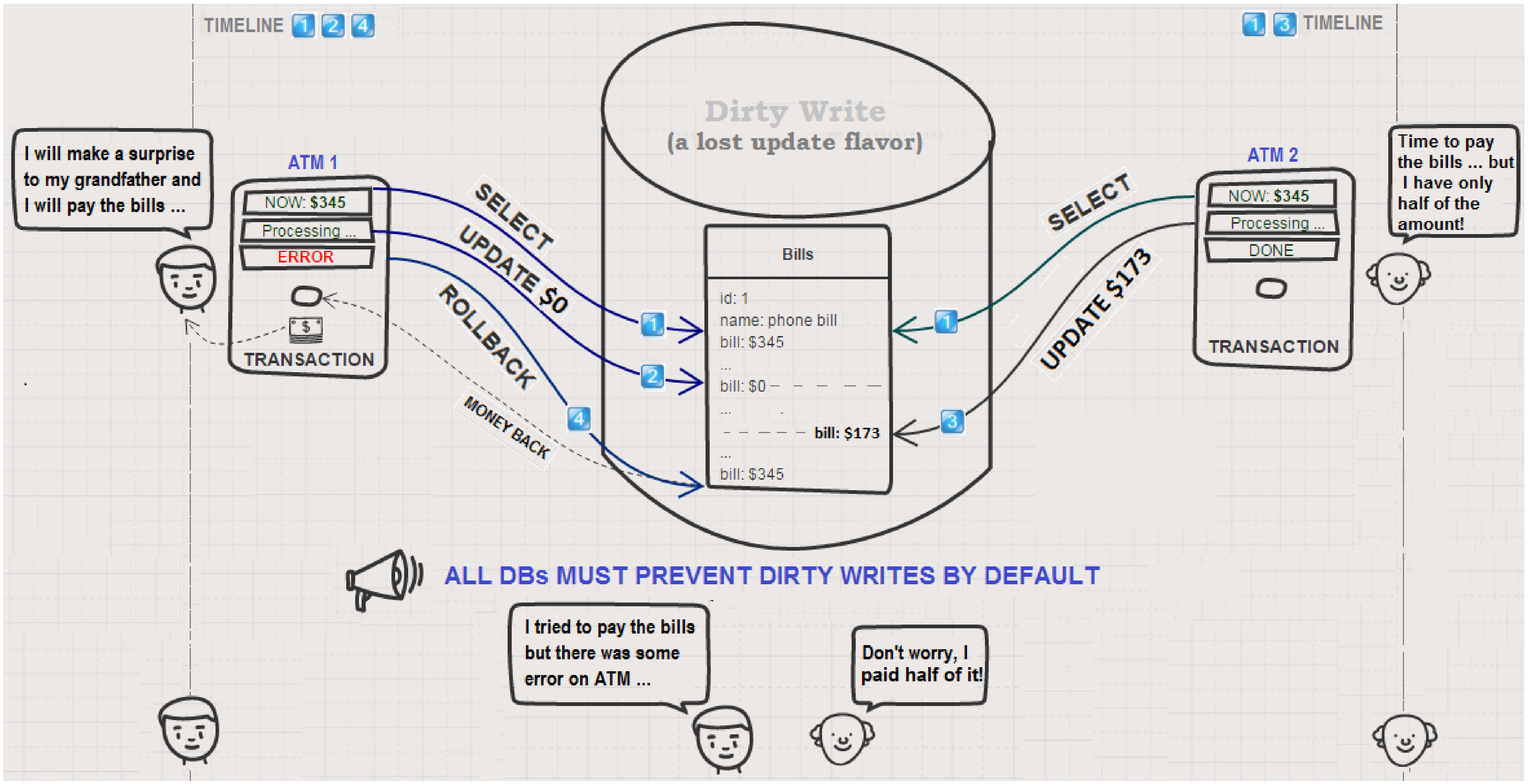
- **This anomaly is specific to Read Uncommitted isolation level.**
- **Mainly, in a dirty-read case, a transaction read the uncommitted modifications of other concurrent transaction that rolls back in the end.**
- **As you can see in the below image, taking a decision based on the uncommitted values can be very frustrated and affects data integrity.**
- **As a quick solution, simply use a higher isolation level.**
- **Always check the default isolation level of your database system (most probably, the default will not be Read Uncommitted, but check it anyway since you must be aware of it).**


TIMELINE 1 4 5 6
TIMELINE 2 3 7
Dirty Read
1 It is time to pay Mark for his work ...
Payer
UPDATE
Mark paycheck (+$5000)
Paycheck
id:1
name:Mark
amount:0
amount:5000
try amount:5200
amount:0
2 It is time to see if my paycheck is ready ...
Mark
SELECT
Mark paycheck amount
3 $5000 nice... I'll withdrawal!
TRANSACTION
Read Uncommited did it again!
$0
Mark
$0
7
ATM
4 Auto-bonus computed for Mark at $200 ...
5 UPDATE
Mark paycheck (+$200)
6 ROLLBACK
TRANSACTION
SELECT
Mark paycheck amount
Use an isolation level higher than Read Uncommitted

Dirty write

The following illustration depicts the dirty-write anomaly:

- **This phenomena is a lost update flavor.**
- **Mainly, in a dirty-write case, a transaction overwrites other concurrent transaction, which means that both transactions are allowed to affect the same row at the same time.**
- **The good news is that, by default, all database systems must prevent dirty writes.**


TIMELINE 1 2 4
1 3 TIMELINE
Dirty Write
(a lost update flavor)
I will make a surprise to my grandfather and I will pay the bills ...
ATM 1
NOW: $345
Processing ...
ERROR
$
TRANSACTION
SELECT
UPDATE $0
ROLLBACK
MONEY BACK
1
2
4
Bills
id: 1
name: phone bill
bill: $345
...
bill: $0
...
bill: $173
...
bill: $345
1
3
SELECT
UPDATE $173
ATM 2
NOW: $345
Processing ...
DONE
TRANSACTION
Time to pay the bills ... but I have only half of the amount!
ALL DBs MUST PREVENT DIRTY WRITES BY DEFAULT
I tried to pay the bills but there was some error on ATM ...
Don't worry, I paid half of it!

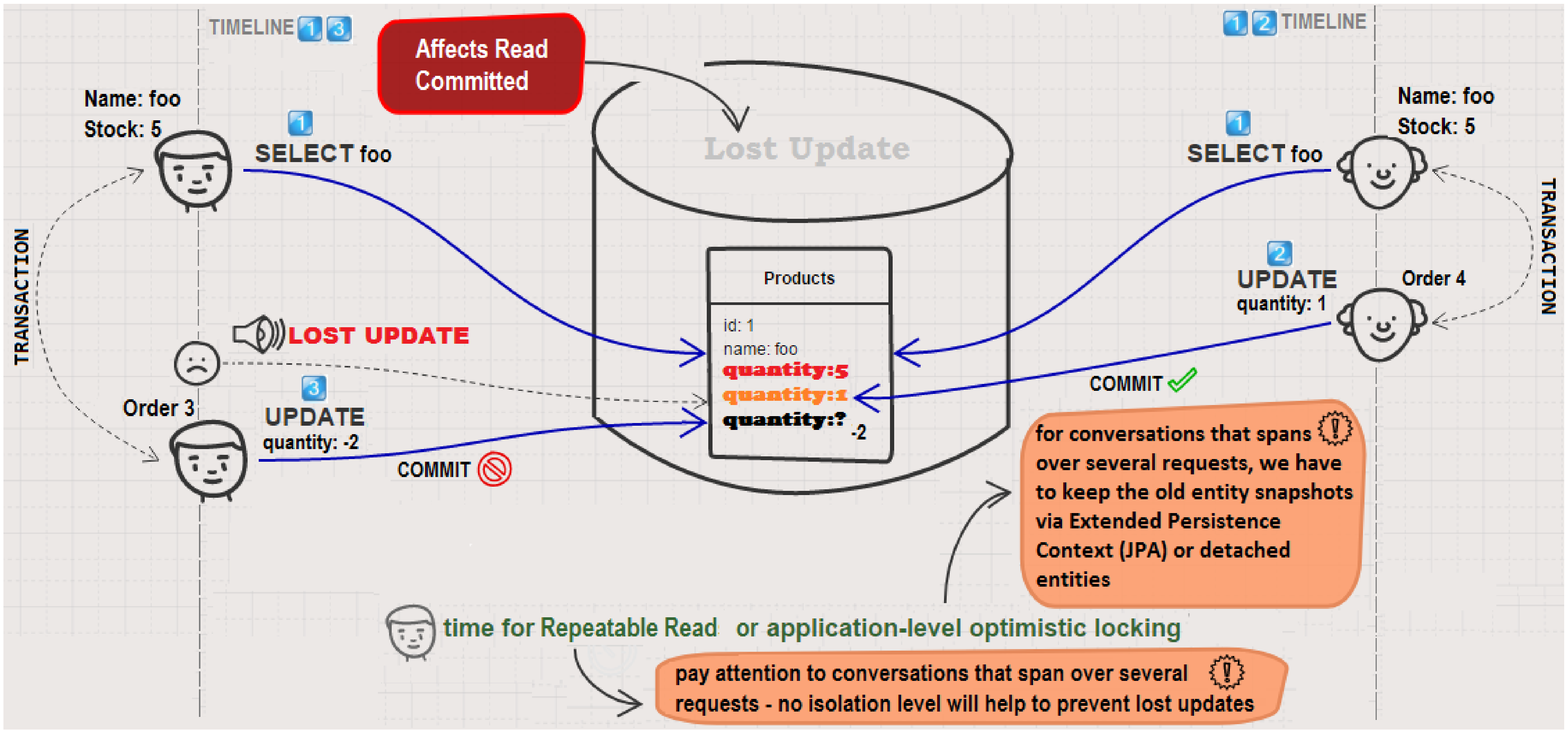
Lost update

The following illustration depicts the lost-update anomaly:

- **In case of this anomaly, a transaction read a row and uses this information to take business decisions (e.g. decisions that may lead to modify that row) without being aware that, in the meanwhile, a concurrent transaction has modified that row as well. When the first transaction commits, it is totally unaware about the update performed by the second transaction. This is a lost update.**

- **This causes data integrity issues (e.g. an e-commerce website inventory can report a negative quantity).**

- **This anomaly affects Read Committed isolation level and can be avoided by setting the Repeatable Reads isolation level (doesn't work for conversations that spans over several requests) or by using application-level optimistic locking (for conversations that spans over several requests, we have to keep the old entity snapshots via Extended Persistence Context (JPA) or detached entities).**


TIMELINE 1 3
Affects Read Committed
1 2 TIMELINE
Lost Update
Name: foo
Stock: 5
1 SELECT foo
Name: foo
Stock: 5
1 SELECT foo
TRANSACTION
TRANSACTION
Products
id: 1
name: foo
quantity:5
quantity:1
quantity:? -2
2 UPDATE quantity: 1
Order 4
COMMIT
LOST UPDATE
Order 3
3 UPDATE quantity: -2
COMMIT
for conversations that spans over several requests, we have to keep the old entity snapshots via Extended Persistence Context (JPA) or detached entities
time for Repeatable Read or application-level optimistic locking
pay attention to conversations that span over several requests - no isolation level will help to prevent lost updates

Phantom read

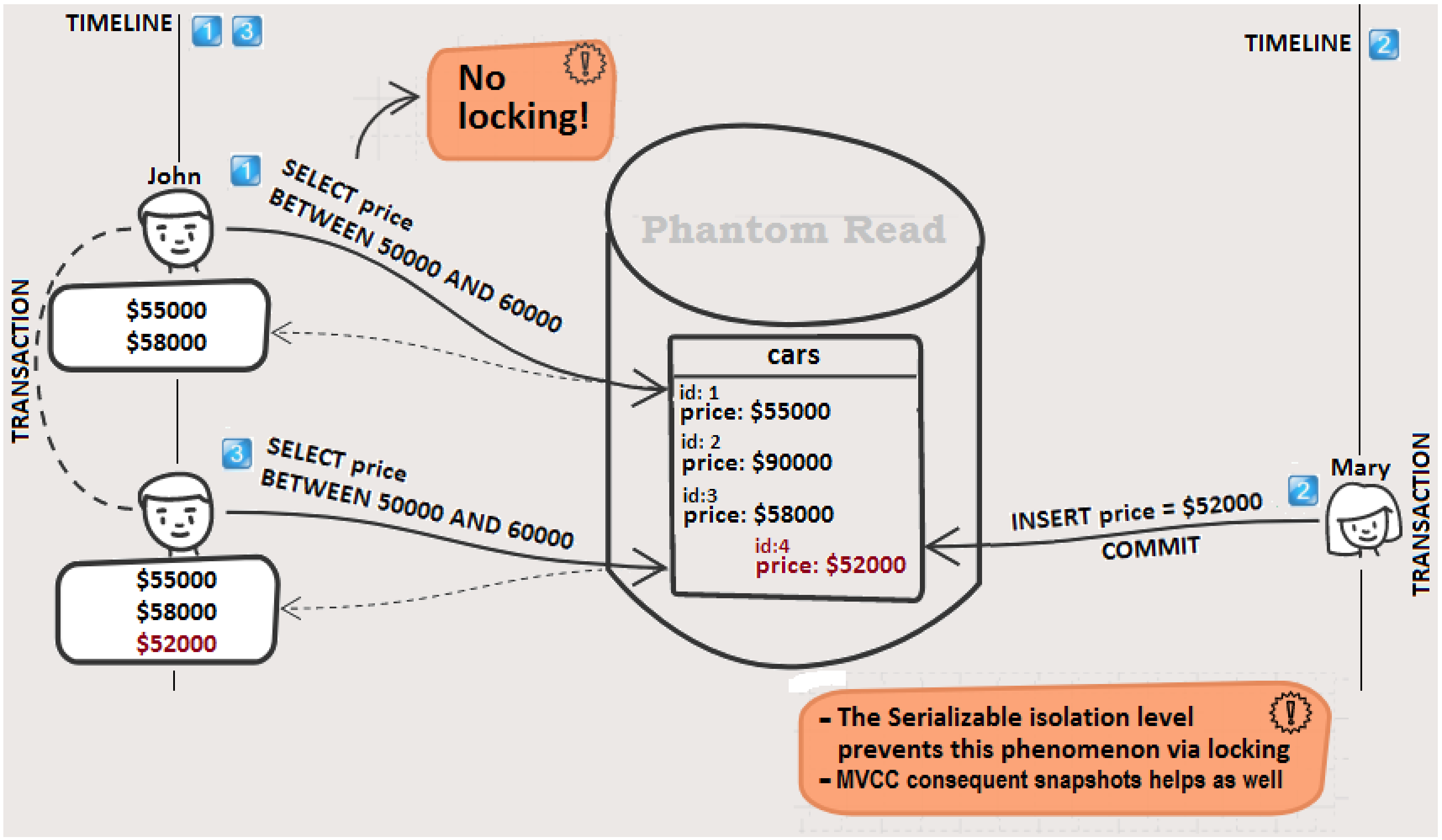
The following illustration depicts the phantom-read anomaly:

- **In a phantom-read, a transaction read a set/range of rows without locking and takes business decisions based on it, while other concurrent transaction can insert (modify) a row that matches this set. The first transaction will not be aware about this insert.**

- **The Serializable isolation level (2PL) prevents this phenomenon via locking.**

- **The MVCC consequent snapshots addresses phantom-reads as well, but there is no locking involved, so a concurrent transaction can still insert (modify) a row.**

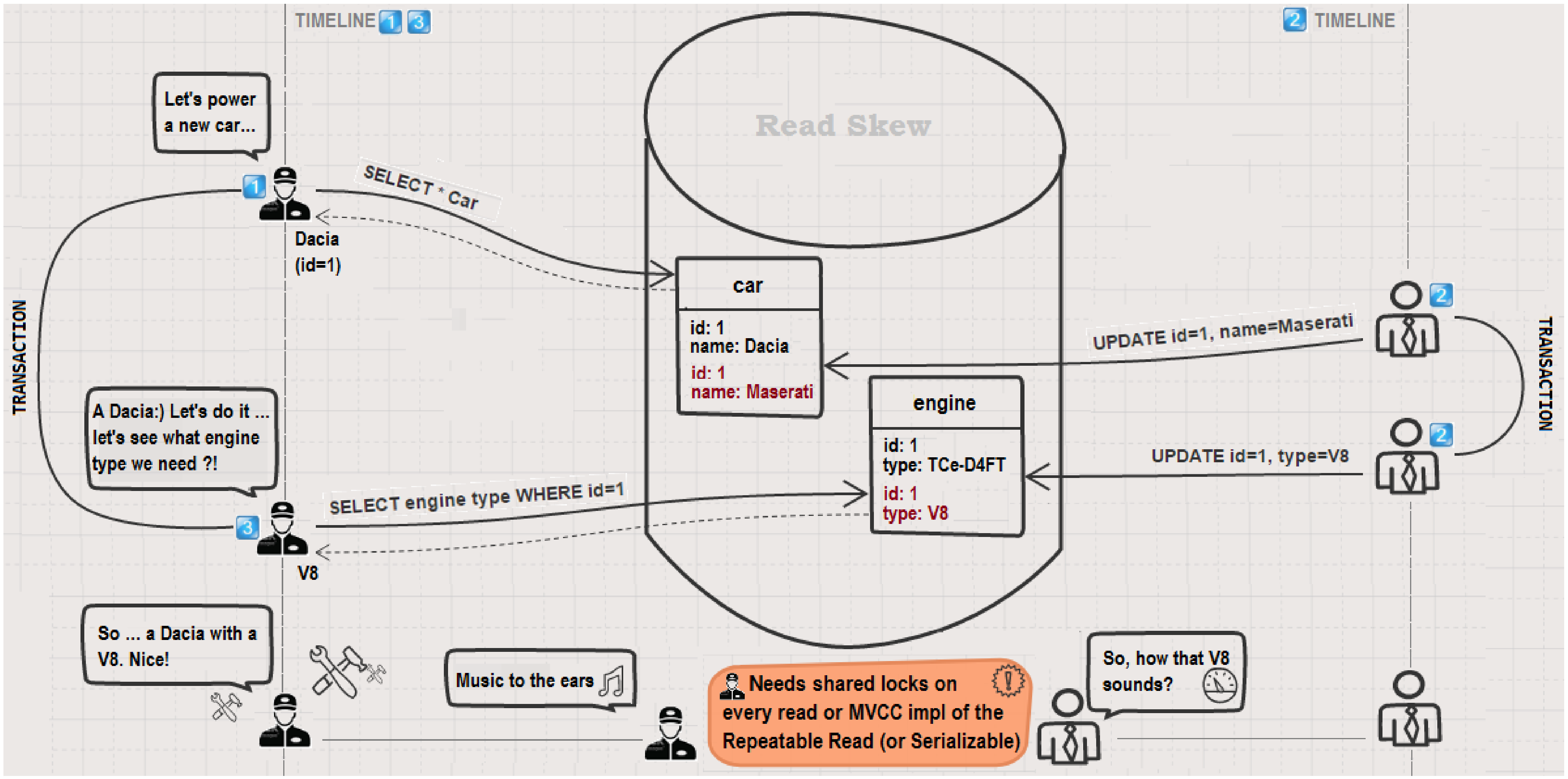
- **The Serializable isolation level and MVCC can produce different results.**


TIMELINE 1 3
No locking!
TIMELINE 2
John
1 SELECT price BETWEEN 50000 AND 60000
Phantom Read
TRANSACTION
$55000
$58000
cars
id: 1
price: $55000
id: 2
price: $90000
id:3
price: $58000
id:4
price: $52000
3 SELECT price BETWEEN 50000 AND 60000
Mary
2
INSERT price = $52000
COMMIT
TRANSACTION
$55000
$58000
$52000
- The Serializable isolation level prevents this phenomenon via locking
- MVCC consequent snapshots helps as well

Read skew

The following illustration depicts the read-skew anomaly:

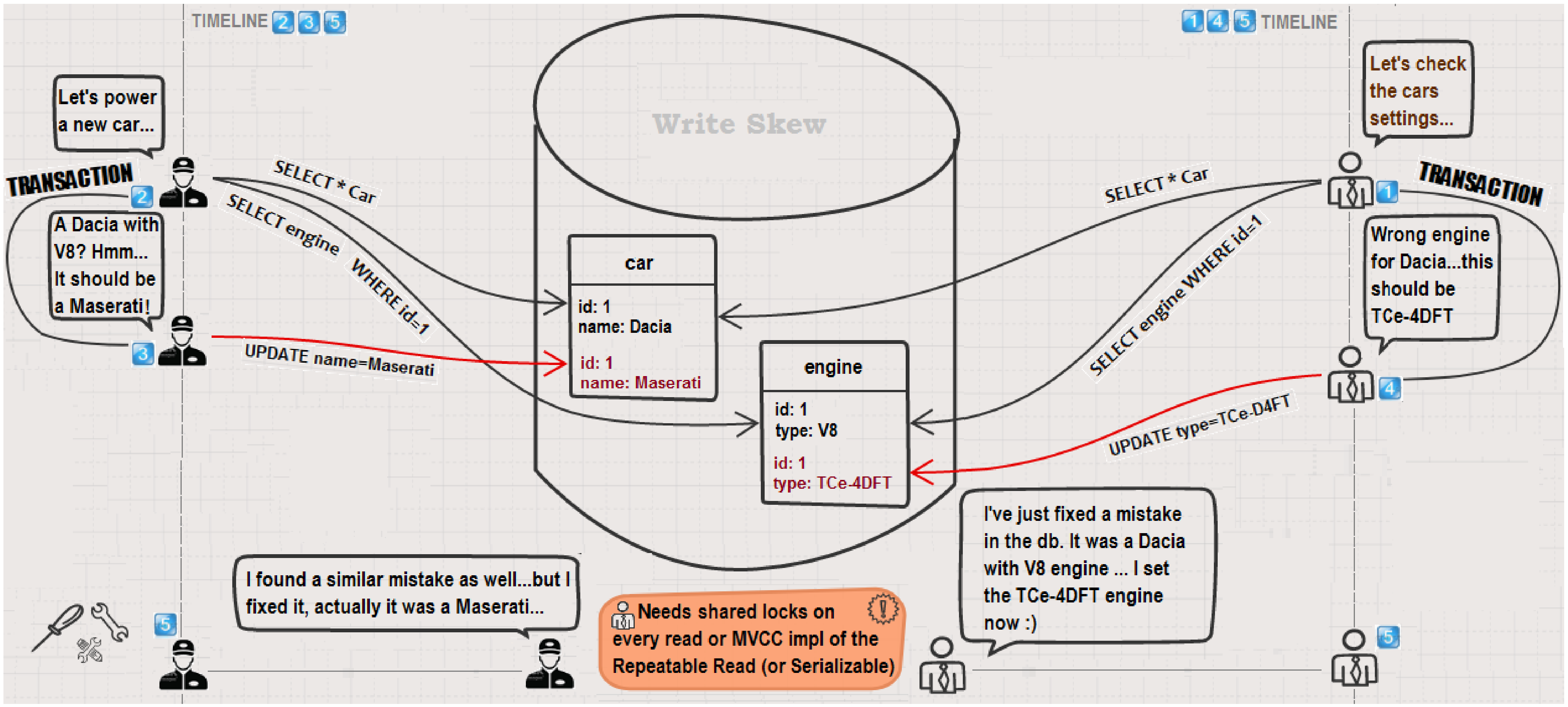
- **Read-skew is an anomaly that can appear when multiple database tables are involved.**
- **Let's consider two tables (e.g., `car` and `engine`).**
- **A transaction reads from the first table (e.g., reads a record from the `car` table).**
- **Further, a concurrent transaction updates the two tables in sync (e.g., updates the car fetched by the first transaction and its corresponding engine).**
- **After both tables are updated, the first transaction reads from the second table (e.g., reads the engine corresponding to the car fetched earlier).**
- **The first transaction sees an older version of the car record and the latest version of the associated engine.**


TIMELINE 1 3
2 TIMELINE
Read Skew
Let's power a new car...
1
SELECT * Car
Dacia (id=1)
car
id: 1
name: Dacia
id: 1
name: Maserati
engine
id: 1
type: TCe-D4FT
id: 1
type: V8
TRANSACTION
UPDATE id=1, name=Maserati
2
UPDATE id=1, type=V8
2
TRANSACTION
A Dacia:) Let's do it ... let's see what engine type we need ?!
SELECT engine type WHERE id=1
3
V8
So ... a Dacia with a V8. Nice!
Music to the ears
Needs shared locks on every read or MVCC impl of the Repeatable Read (or Serializable)
So, how that V8 sounds?

Write skew

The following illustration depicts the write-skew anomaly:

- **Write-skew is an anomaly that can appear when multiple database tables are involved.**
- **Let's consider two tables (e.g., `car` and `engine`).**
- **Both tables should be updated in sync, but write-skew allows to concurrent transactions to break this constraint.**

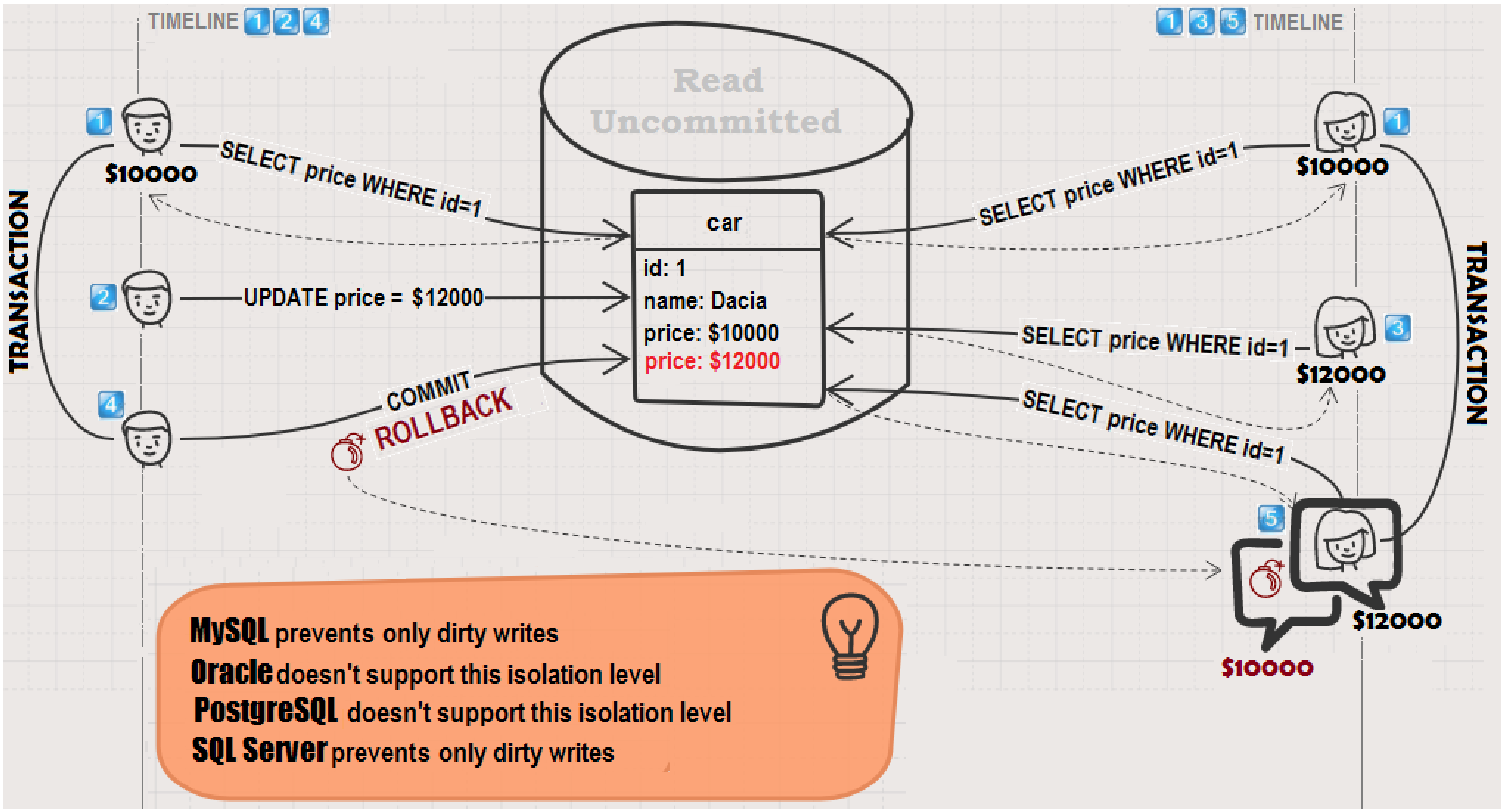

TIMELINE 2 3 5
1 4 5 TIMELINE
Write Skew
Let's power a new car...
TRANSACTION
2
A Dacia with V8? Hmm... It should be a Maserati!
3
SELECT * Car
SELECT engine WHERE id=1
UPDATE name=Maserati
car
id: 1
name: Dacia
id: 1
name: Maserati
engine
id: 1
type: V8
id: 1
type: TCe-4DFT
Let's check the cars settings...
TRANSACTION
1
SELECT * Car
SELECT engine WHERE id=1
Wrong engine for Dacia...this should be TCe-4DFT
4
UPDATE type=TCe-D4FT
I've just fixed a mistake in the db. It was a Dacia with V8 engine ... I set the TCe-4DFT engine now :)
5
I found a similar mistake as well...but I fixed it, actually it was a Maserati...
Needs shared locks on every read or MVCC impl of the Repeatable Read (or Serializable)

Read Uncommitted

The following illustration depicts the Read Uncommitted isolation level:

- **In MySQL and SQL Server, this isolation level provides protection only against dirty writes.**

- **In PostgreSQL and Oracle, this isolation level is not supported (default is Read Committed)**



MySQL prevents only dirty writes

Oracle doesn't support this isolation level

PostgreSQL doesn't support this isolation level

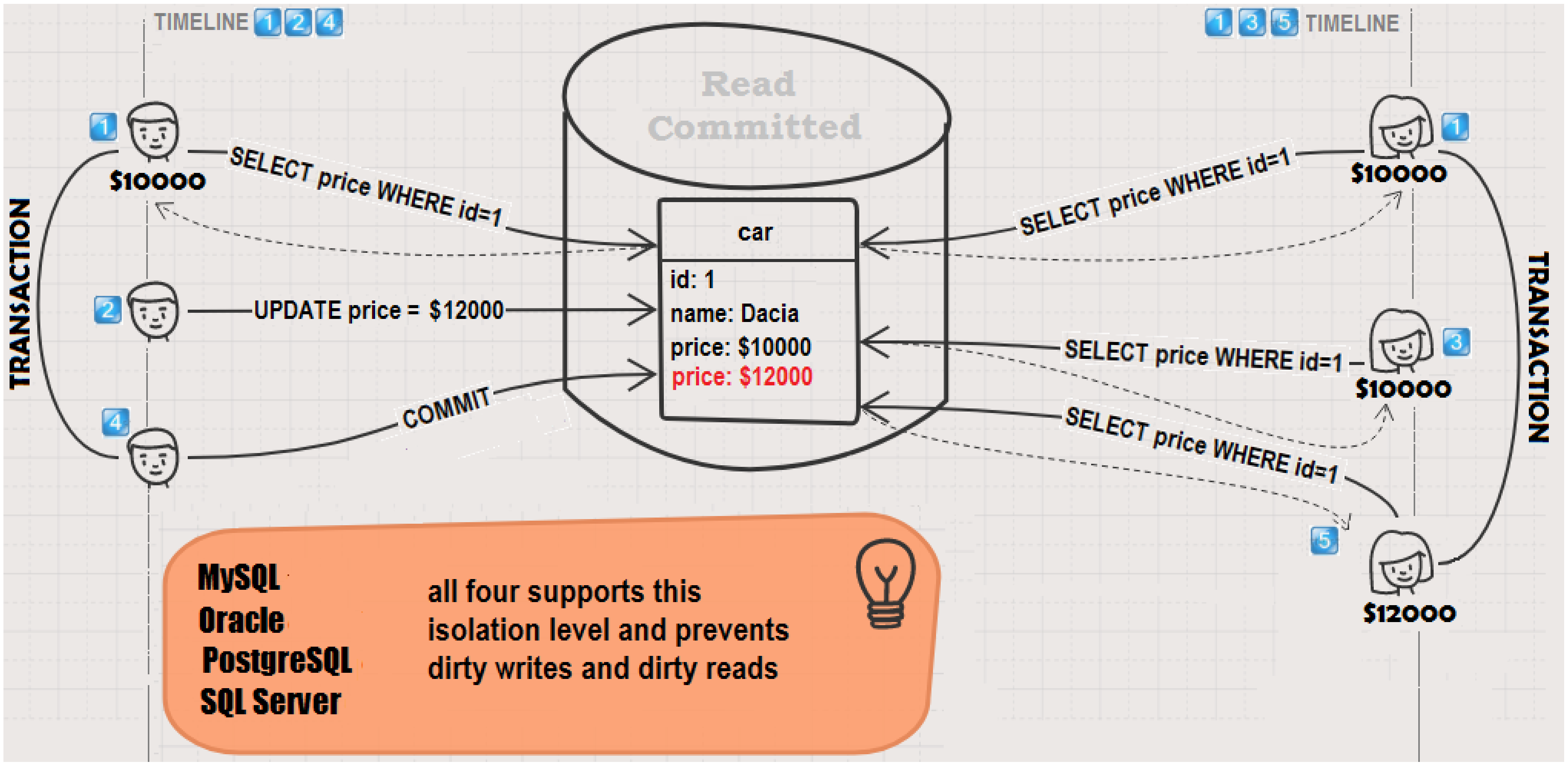
SQL Server prevents only dirty writes

Read Committed

The following illustration depicts the Read Committed isolation level:

- **This isolation level is supported by many RDBMSs.**

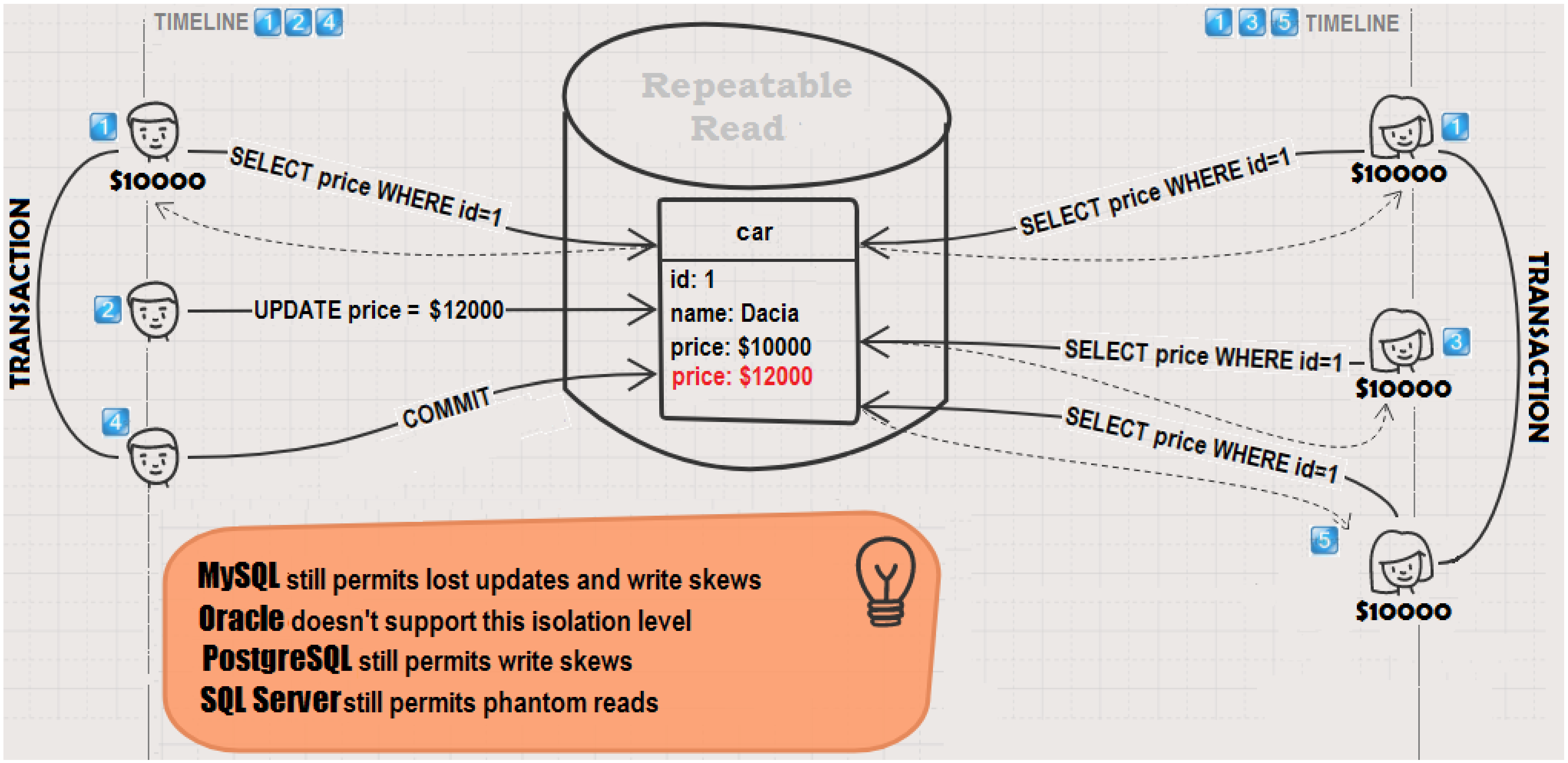
- **It is a common approach to have this isolation level as default, but pay attention that even if it prevents dirty-writes and dirty-reads, it still leaves the gate open to many other anomalies.**


TIMELINE 1 2 4
1 3 5 TIMELINE
Read Committed
TRANSACTION
TRANSACTION
1
$10000
SELECT price WHERE id=1
2
UPDATE price = $12000
4
COMMIT
car
id: 1
name: Dacia
price: $10000
price: $12000
SELECT price WHERE id=1
1
$10000
SELECT price WHERE id=1
3
$10000
SELECT price WHERE id=1
5
$12000
MySQL
Oracle
PostgreSQL
SQL Server
all four supports this isolation level and prevents dirty writes and dirty reads

Repeatable Read

The following illustration depicts the Repeatable Read isolation level:

- **In Oracle, this isolation level is not supported**
- **In PostgreSQL, this isolation level still permits write-skews**
- **In MySQL, it still permits lost-updates and write-skews**
- **In SQL Server, it still permits phantom-reads**

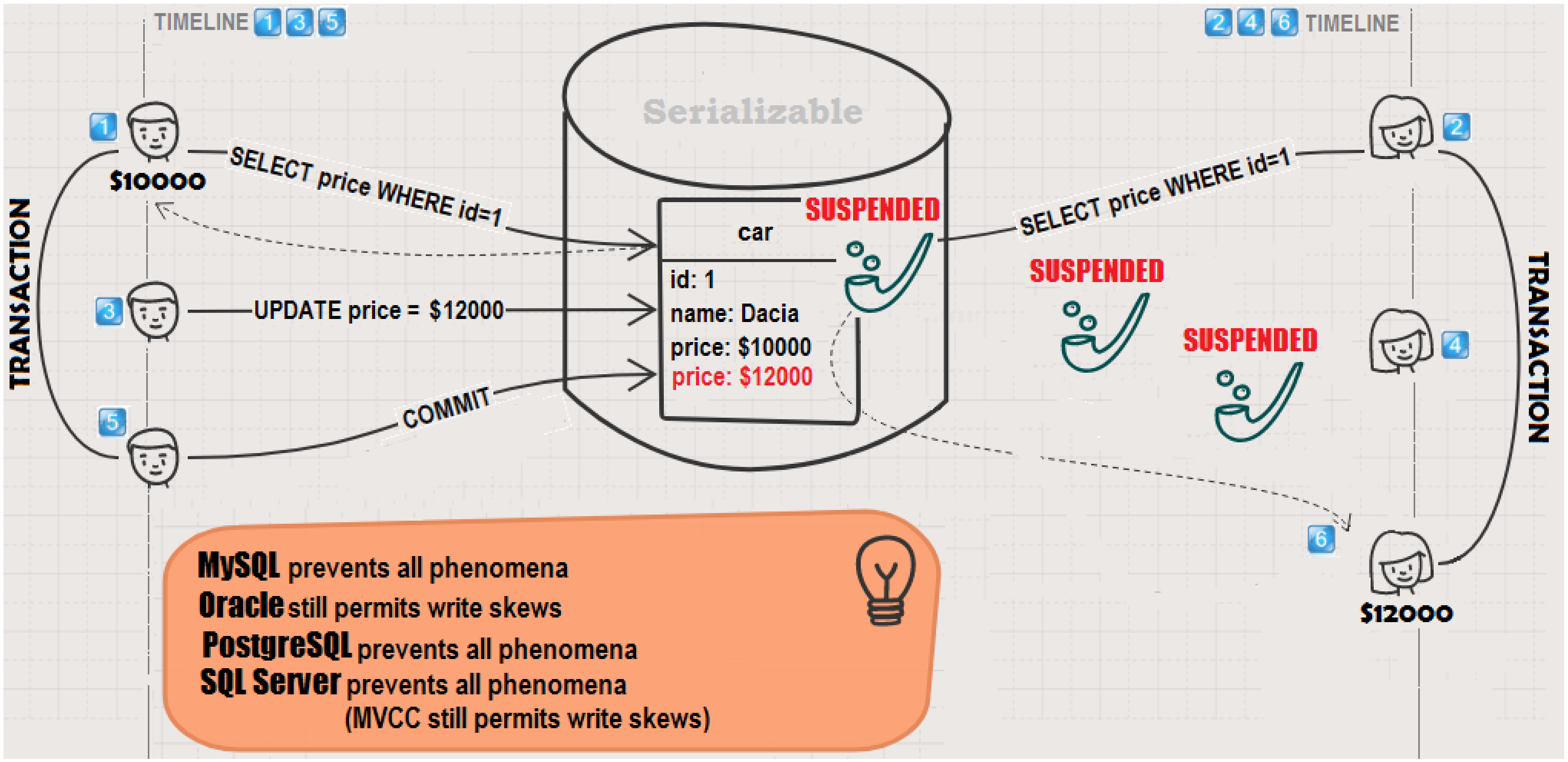


MySQL still permits lost updates and write skews
Oracle doesn't support this isolation level
PostgreSQL still permits write skews
SQL Server still permits phantom reads

Serializable

The following illustration depicts the Serializable isolation level:

- **In Oracle, this isolation level still permits write skews.**
- **MySQL, SQL Server and PostgreSQL prevents all phenomena. SQL Server MVCC still permits write skews.**



MySQL prevents all phenomena
Oracle still permits write skews
PostgreSQL prevents all phenomena
SQL Server prevents all phenomena
(MVCC still permits write skews)

Part III

JPA & Hibernate

Identifiers

The hi/lo algorithm

Choose between identifiers generation strategies

The equals() and hashCode() methods

In MySQL & Hibernate 5 Avoid AUTO Generator Type

Hibernate natural id

The hi/lo algorithm

The following illustration depicts the *hi/lo* algorithm:

- **By default, the `SEQUENCE` generator must hit the database for each new sequence value via a `SELECT` statement.**

- **The *hi/lo* algorithm is an optimization algorithm for generating sequences of identifiers in-memory.**

- **This algorithm reduce the database roundtrips by using a simple formula based on a configurable increment (*inc*) and a starting value (*hi*). This increment gives the number of identifiers that can be generated in-memory (aka, number of *lo* entries). Mainly, in a single database roundtrip, the *hi/lo* algorithm fetches from the database a *hi* value. Using this *hi* value, Hibernate generates in-memory a number of identifiers equal to the value of *inc* increment.**

- **The *hi* value can be provided by the database sequence or the table generator.**

- **This algorithm splits the sequences domain into synchronously *hi* groups.**

- **Identifiers range is given by the following formula:**

[inc x (hi - 1) + 1, inc x hi]
(e.g., inc = 4, hi =1, range: [1, 4])

- **While *lo* is in this range, no database roundtrips for fetching identifiers are needed and identifiers can be safely used.**

- **When all *lo* values are used, a new *hi* value is fetched via a new database roundtrip.**

- **The *hi/lo* algorithm is not the proper choice in the presence of external systems that needs to insert in the tables that uses *hi/lo*. Because the database sequence is not aware about the highest in-memory generated identifier it returns sequence values that might be already used as identifiers. This leads to duplicate identifiers errors.**

- **For such cases, rely on *pooled* and *pooled-lo* algorithms. These are optimizations of *hi/lo* meant to allow external systems to work as expected.**

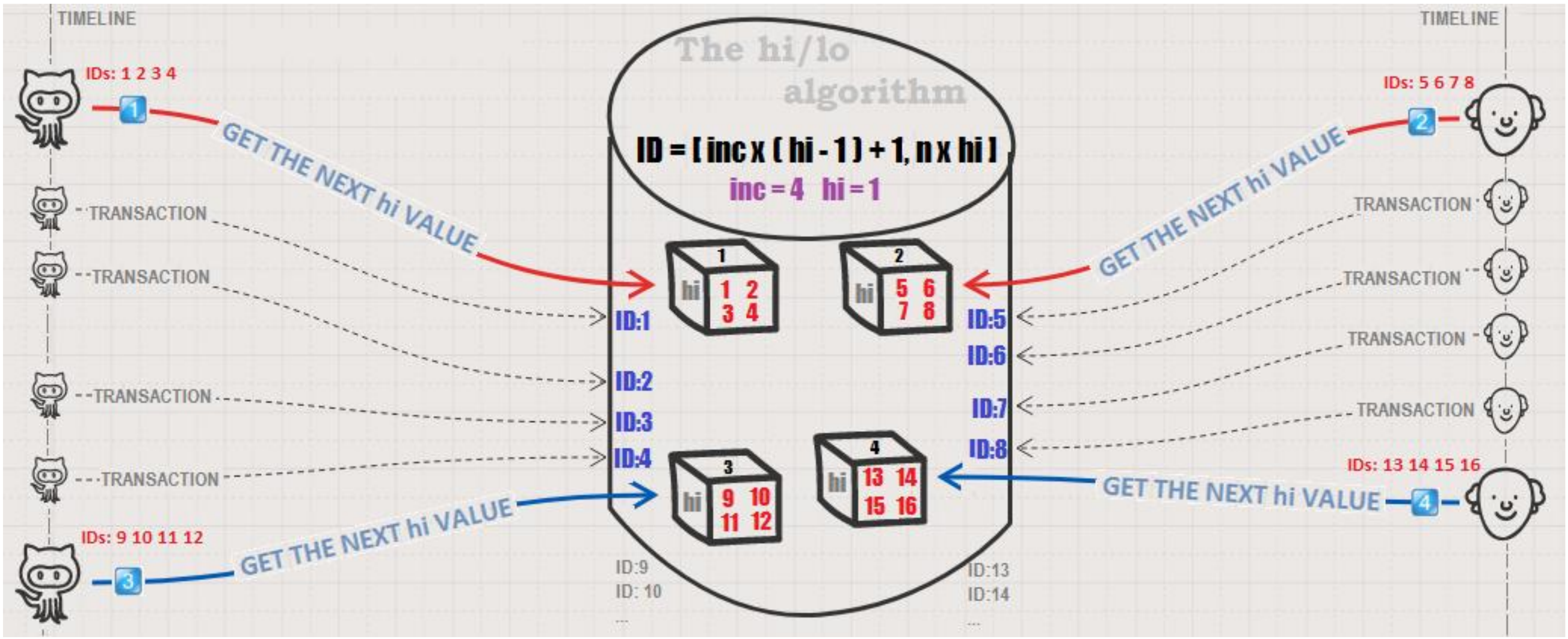
Example:

```
@Id
@GeneratedValue(
    strategy = GenerationType.SEQUENCE,
    generator = "hilo"
)
@GenericGenerator(
    name = "hilo",
    strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
    parameters = {
        @Parameter(name = "sequence_name", value = "hilo"),
        @Parameter(name = "initial_value", value = "1"),
        @Parameter(name = "increment_size", value = "4"),
        @Parameter(name = "optimizer", value = "hilo")
    }
)

private Long id;
```

Applications:

- **How To Generate Sequences Of Identifiers Via Hibernate hi/lo Algorithm**
- **Hibernate hi/lo Algorithm And External Systems Issue**
- **How To Generate Sequences Of Identifiers Via Hibernate pooled Algorithm**
- **How To Generate Sequences Of Identifiers Via Hibernate pooled-lo Algorithm**



TIMELINE
TIMELINE
The hi/lo algorithm
ID = [inc x (hi - 1) + 1, n x hi]
inc = 4 hi = 1
IDs: 1 2 3 4
IDs: 5 6 7 8
IDs: 9 10 11 12
IDs: 13 14 15 16
GET THE NEXT hi VALUE
TRANSACTION
ID:1
ID:2
ID:3
ID:4
ID:5
ID:6
ID:7
ID:8
ID:9
ID: 10
ID:13
ID:14

Choose between identifiers generation strategies

The following illustration depicts how to choose between identifiers generation strategies:

- **Identifiers (PKs) can be explicitly (manually) set or generated by the database server or by the application (data access layer).**

- **We can rely on natural primary keys (with real meaning in our world - preferable numerical) or surrogate primary keys (the current row data is not relevant).**

- **Surrogate primary keys can be numerical or GUID/UUID.**

- **From the perspective of performance (and not only), the numerical identifiers are recommended against GUID/UUID because they have a smaller footprint on memory and indexing process.**

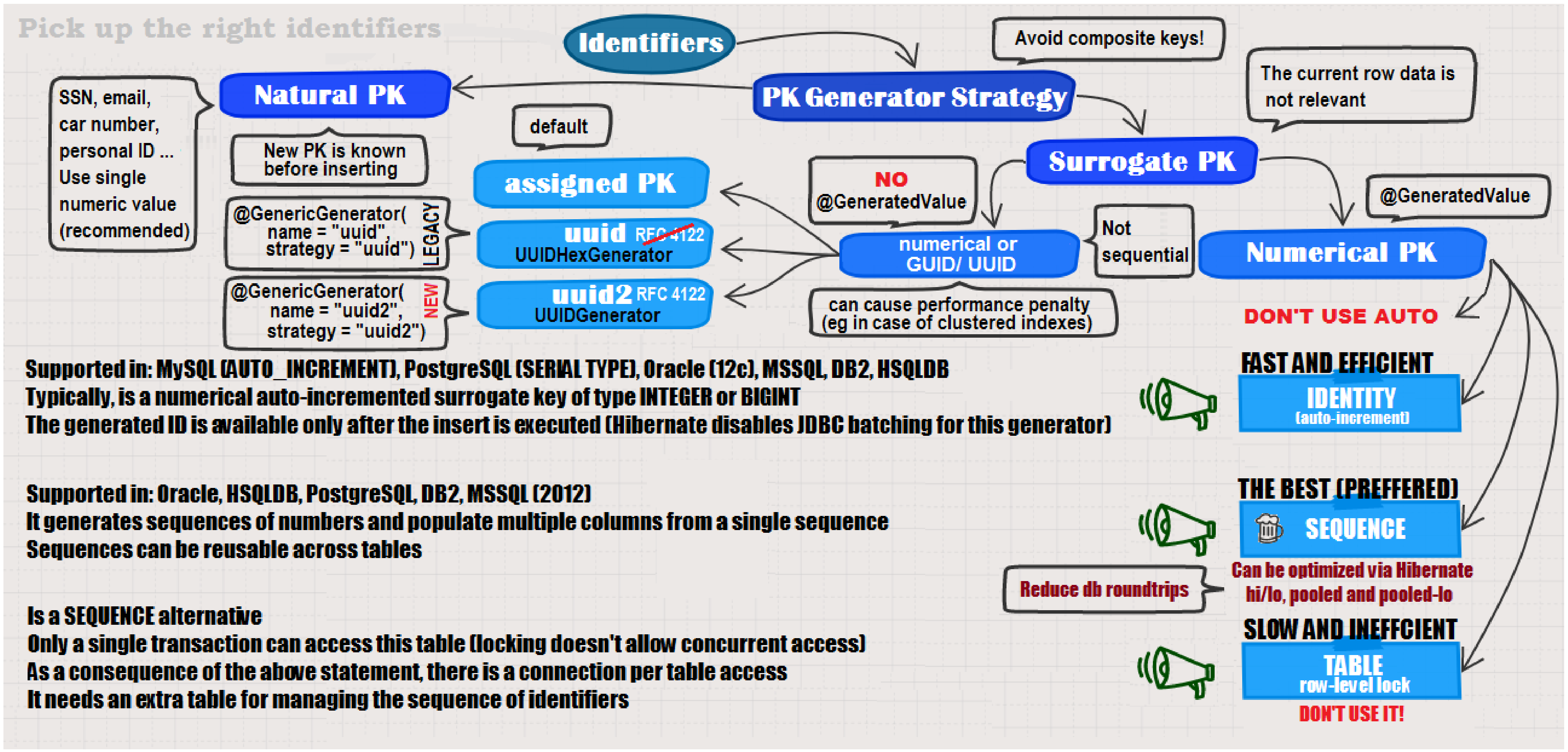
- **From the GUID/UUID generators, the RFC 4122 compliant (`UUIDGenerator`) is recommended.**

- **From the numerical generators, the `SEQUENCE` generator is the best choice and it can take advantage of Hibernate optimization algorithms such as *hi/lo*, *pooled* and *pooled-lo*.**

- **From the numerical generators, the `TABLE` generator is the worst choice from the performance perspective.**

- **Avoid composite keys as much as possible because they increase the footprint on memory and indexing process.**

- **No matter what kind of identifiers you choose, keep in mind to ensure their uniqueness in your system.**


Pick up the right identifiers
Identifiers
PK Generator Strategy
Avoid composite keys!
Natural PK
SSN, email, car number, personal ID ... Use single numeric value (recommended)
New PK is known before inserting
default
assigned PK
@GenericGenerator(name = "uuid", strategy = "uuid")
LEGACY
uuid ~~RFC 4122~~
UUIDHexGenerator
@GenericGenerator(name = "uuid2", strategy = "uuid2")
NEW
uuid2 RFC 4122
UUIDGenerator
NO @GeneratedValue
numerical or GUID/ UUID
can cause performance penalty (eg in case of clustered indexes)
Not sequential
Surrogate PK
The current row data is not relevant
@GeneratedValue
Numerical PK
DON'T USE AUTO
FAST AND EFFICIENT
IDENTITY (auto-increment)
THE BEST (PREFFERED)
SEQUENCE
Can be optimized via Hibernate hi/lo, pooled and pooled-lo
Reduce db roundtrips
SLOW AND INEFFCIENT
TABLE row-level lock
DON'T USE IT!

Supported in: MySQL (AUTO_INCREMENT), PostgreSQL (SERIAL TYPE), Oracle (12c), MSSQL, DB2, HSQLDB
Typically, is a numerical auto-incremented surrogate key of type INTEGER or BIGINT
The generated ID is available only after the insert is executed (Hibernate disables JDBC batching for this generator)

Supported in: Oracle, HSQLDB, PostgreSQL, DB2, MSSQL (2012)
It generates sequences of numbers and populate multiple columns from a single sequence
Sequences can be reusable across tables

Is a SEQUENCE alternative
Only a single transaction can access this table (locking doesn't allow concurrent access)
As a consequence of the above statement, there is a connection per table access
It needs an extra table for managing the sequence of identifiers

The equals() and hashCode() methods

The following illustrations depicts how to write `equals()` and `hashCode()`:

- **Any Java developer is familiar with the `equals()` and `hashCode()` methods, but not any Java developer knows how to correctly override these methods from Hibernate perspective.**

- **The main statement that we need to consider is the fact that Hibernate requires that *an entity is equal to itself across all its state transitions* (*transient*, *attached*, *detached* and *removed*).**

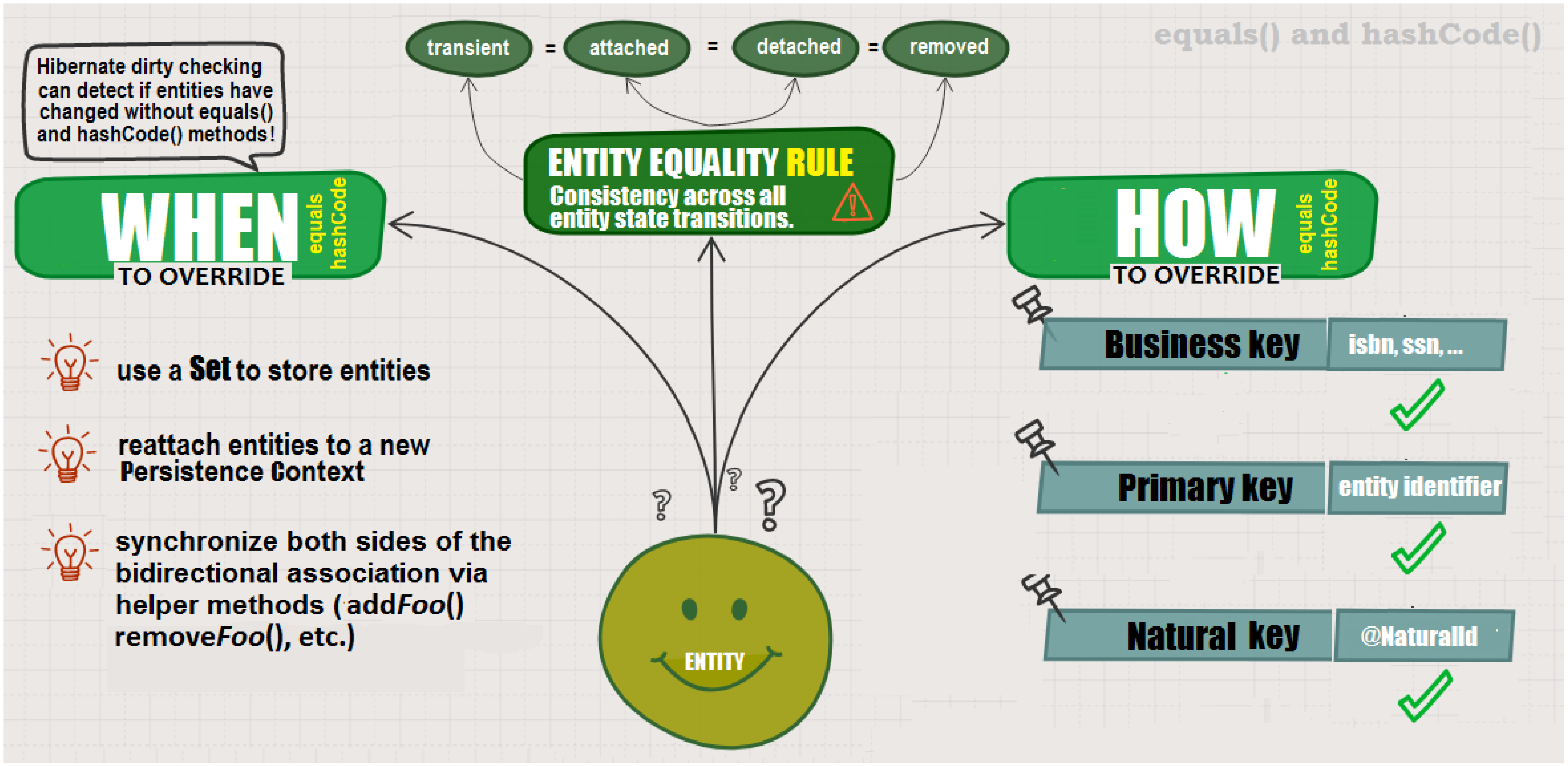
- **In order to detect the entities changes, Hibernate uses its internal mechanism known as *dirty checking*. This mechanism doesn't uses `equals()` and `hashCode()`.**

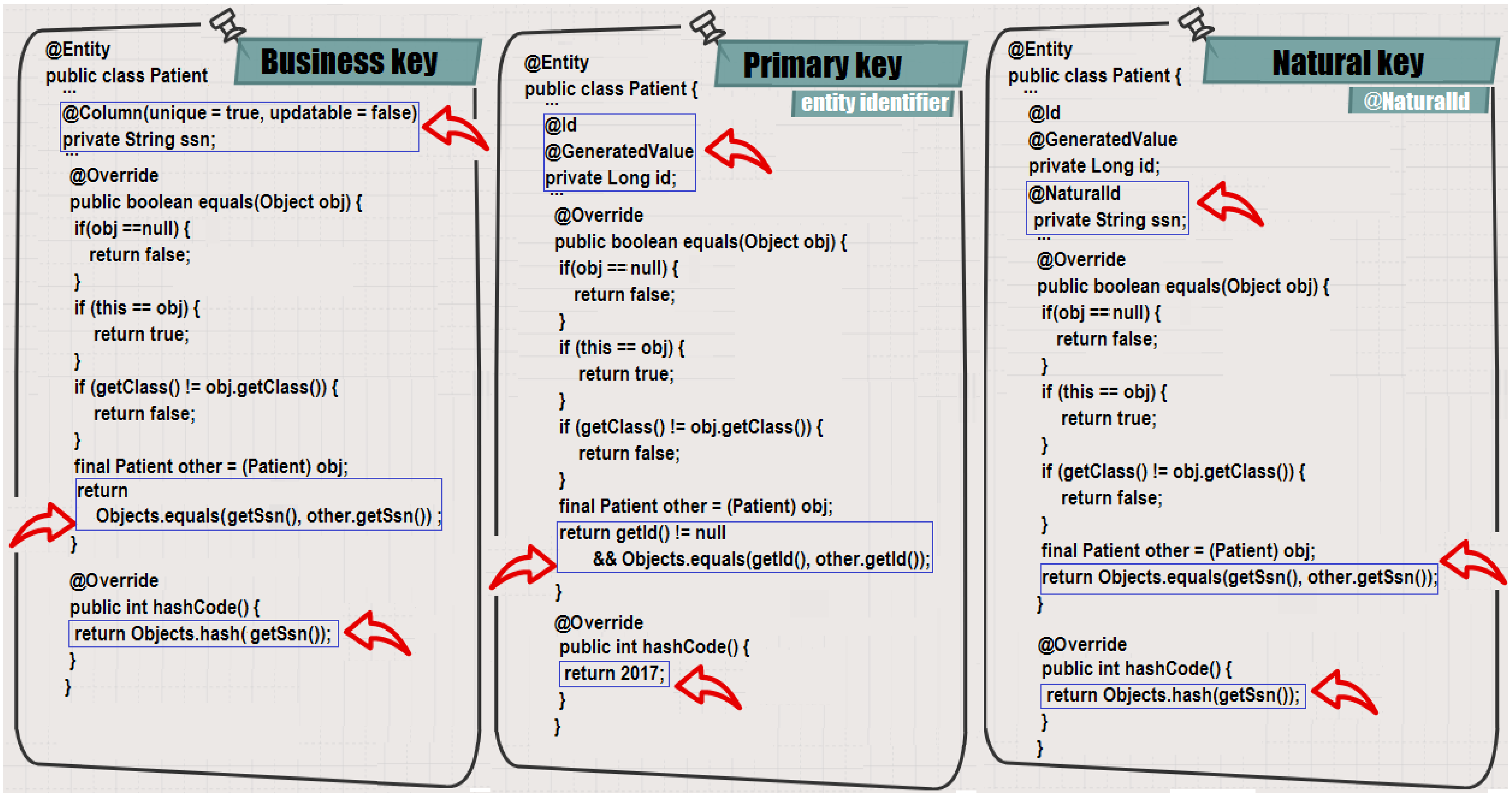
- **But, conforming to Hibernate documentation, if we store entities in a `Set` or we need to reattach entities to a new Persistence Context then we need to override `equals()` and `hashCode()` as well.**

- **In order to implement `equals()` and `hashCode()` and respect the consistency of entity equality across all its state transitions, we need to be aware of several cases:**
  - **rely on a business key (an unique and non-updatable field that it is not the identifier of the entity) - this is recommended;**
  - **rely on Hibernate, `@NaturalId;`**
  - **rely on the generated entity identifier, but pay extra attention to the `hashCode()` implementation and to the *transient* state;**
  - **avoid Lombok, `@EqualsAndHashCode` and `@Data;`**
  - **Don't rely on default `equals()` and `hashCode()`!**

Application:

- **Why To Avoid Lombok @EqualsAndHashCode And @Data In Entities And How To Override equals() And hashCode()**


equals() and hashCode()
Hibernate dirty checking can detect if entities have changed without equals() and hashCode() methods!
transient = attached = detached = removed
ENTITY EQUALITY RULE
Consistency across all entity state transitions.
WHEN equals hashCode TO OVERRIDE
use a Set to store entities
reattach entities to a new Persistence Context
synchronize both sides of the bidirectional association via helper methods (addFoo() removeFoo(), etc.)
ENTITY
HOW equals hashCode TO OVERRIDE
Business key
isbn, ssn, ...
Primary key
entity identifier
Natural key
@NaturalId



```
@Entity
public class Patient
 ...
  @Column(unique = true, updatable = false)
  private String ssn;
  ...
   @Override
   public boolean equals(Object obj) {
    if(obj ==null) {
      return false;
    }
    if (this == obj) {
      return true;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Patient other = (Patient) obj;
    return
      Objects.equals(getSsn(), other.getSsn()) ;
   }

   @Override
   public int hashCode() {
    return Objects.hash( getSsn());
   }
  }
```

```
@Entity
public class Patient {
 ...
  @Id
  @GeneratedValue
  private Long id;
  ...
   @Override
   public boolean equals(Object obj) {
    if(obj == null) {
      return false;
    }
    if (this == obj) {
      return true;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Patient other = (Patient) obj;
    return getId() != null
        && Objects.equals(getId(), other.getId());
   }
   @Override
    public int hashCode() {
    return 2017;
    }
   }
```

```
@Entity
public class Patient {
 ...
  @Id
  @GeneratedValue
  private Long id;
  @NaturalId
  private String ssn;
  ...
   @Override
   public boolean equals(Object obj) {
    if(obj == null) {
      return false;
    }
    if (this == obj) {
      return true;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Patient other = (Patient) obj;
    return Objects.equals(getSsn(), other.getSsn());
   }

   @Override
    public int hashCode() {
    return Objects.hash(getSsn());
    }
   }
```

In MySQL & Hibernate 5 Avoid AUTO Generator Type

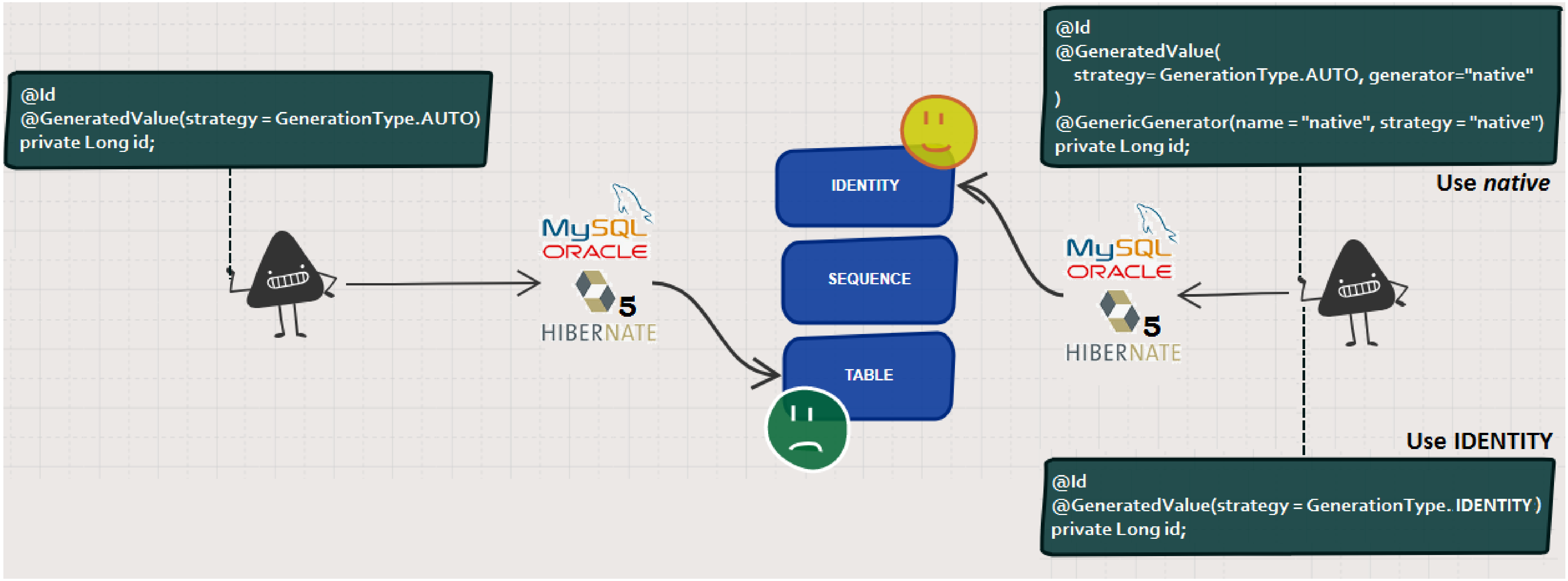
The following illustrations depicts why we should avoid the `AUTO` generator type in MySQL and Hibernate 5:

- **Using MySQL with Hibernate 5 and `AUTO` generator type is a performance penalty because Hibernate 5 will decide to use the `TABLE` generator type, which has serious performance issues.**

- **The `TABLE` generator type doesn't scale well and is much more slower than `IDENTITY` and `SEQUENCE` generators types even for a single database connection.**

Application:

- **MySQL & Hibernate 5 Avoid AUTO Generator Type**



```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

```
@Id
@GeneratedValue(
    strategy= GenerationType.AUTO, generator="native"
)
@GenericGenerator(name = "native", strategy = "native")
private Long id;
```

```
@Id
@GeneratedValue(strategy = GenerationType. IDENTITY )
private Long id;
```

Hibernate natural id (@NaturalId)

The following illustration depicts the usage of Hibernate `@NaturalId`:

- **Hibernate provides support for declaring a *business key* as a natural id via the `@NaturalId` annotation.**

- **The *business key(s)* must be unique (e.g., book ISBN, people SSN, etc).**

- **An entity can have in the same time a primary key (e.g., auto-generated) and one or more natural ids as well.**

- **When there are more (multiple fields are annotated with `@NaturalId`) you have to perform the find operation by specifying all of them otherwise you will get an exception of type: *Entity [...] defines its natural-id with n properties but only k were specified.***

- **If your entity has a single `@NaturalId`, you can find it via the `Session#bySimpleNaturalId()` methods.**

- **If your entity has a more than one `@NaturalId`, you can find it via the `Session#byNaturalId()` methods.**

- **Natural ids can be *mutable* or *immutable* (default). You can easily switch to *mutable* by writing: `@NaturalId(mutable = true)`**

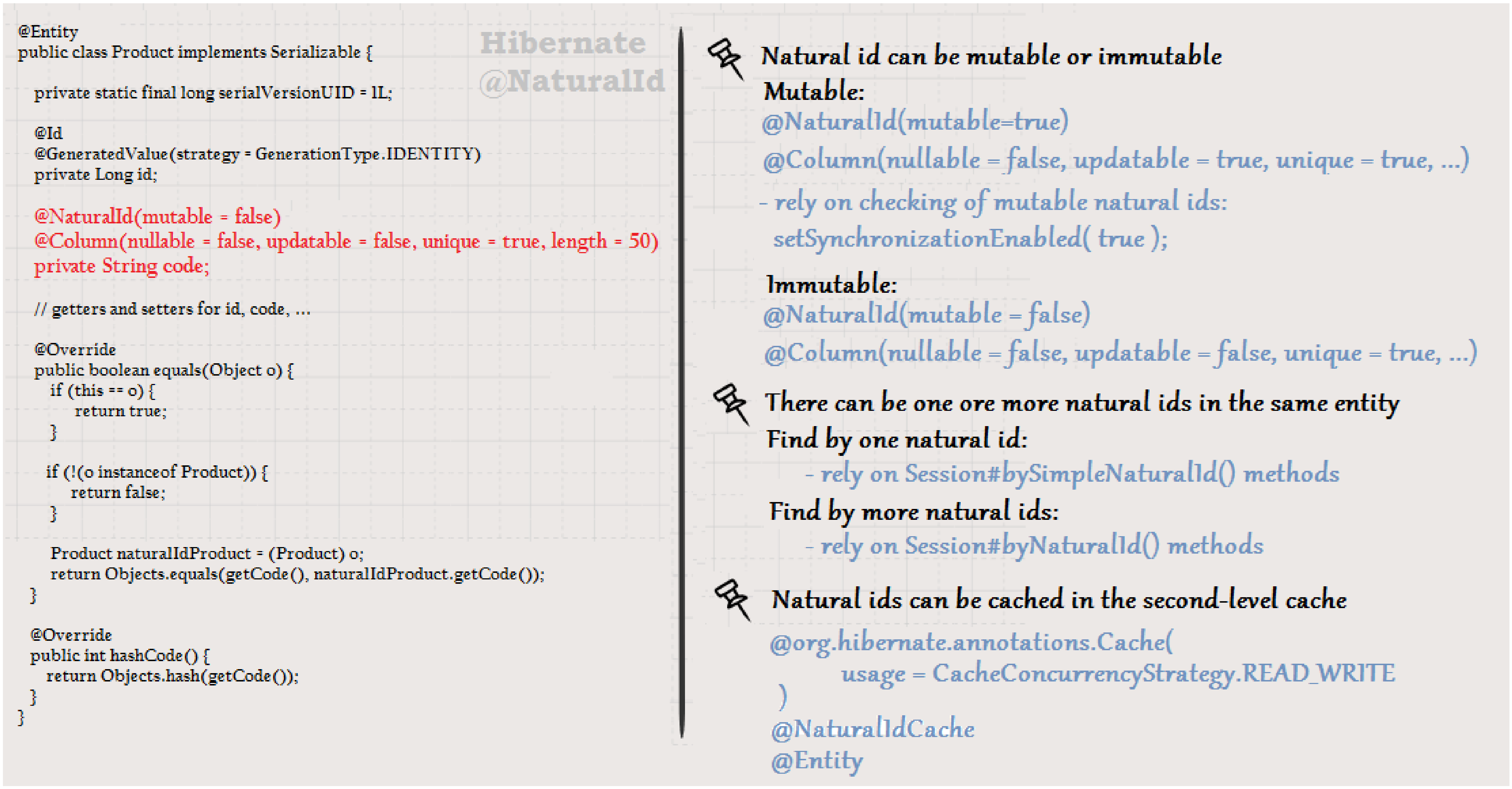
- **It is advisable that a field marked as `@NaturalId` to be mark with `@Column` as well, most commonly like this (*immutable*): `@Column(nullable = false, updatable = false, unique = true)`. If you natural id is *mutable*, set `updatable=true`.**

- The `equals()` and `hashCode()` methods should be implemented to be natural id centric, as in the example from the below picture.

- If the Second Level Cache is used that you can easily cache natural ids as well via `@NaturalIdCache`.

Applications:

- **How To Use Hibernate @NaturalId in SpringBoot**
- **How To Use Hibernate @NaturalId In Spring Boot Style**
- **How To Use Hibernate @NaturalIdCache For Skipping The Entity Identifier Retrieval**
- **How To Define An Association That Reference @NaturalId**



Hibernate @NaturalId

```
@Entity
public class Product implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @NaturalId(mutable = false)
  @Column(nullable = false, updatable = false, unique = true, length = 50)
  private String code;

  // getters and setters for id, code, ...

  @Override
  public boolean equals(Object o) {
    if (this == o) {
        return true;
    }

    if (!(o instanceof Product)) {
        return false;
    }

    Product naturalIdProduct = (Product) o;
    return Objects.equals(getCode(), naturalIdProduct.getCode());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getCode());
  }
}
```

Natural id can be mutable or immutable

Mutable:

```
@NaturalId(mutable=true)
@Column(nullable = false, updatable = true, unique = true, ...)
```

- rely on checking of mutable natural ids:
  `setSynchronizationEnabled( true );`

Immutable:

```
@NaturalId(mutable = false)
@Column(nullable = false, updatable = false, unique = true, ...)
```

There can be one ore more natural ids in the same entity

Find by one natural id:

- rely on Session#bySimpleNaturalId() methods

Find by more natural ids:

- rely on Session#byNaturalId() methods

Natural ids can be cached in the second-level cache

```
@org.hibernate.annotations.Cache(
        usage = CacheConcurrencyStrategy.READ_WRITE
 )
@NaturalIdCache
@Entity
```

The end!

Thank you!