

ACID

**Atomicity**

**All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.**

**For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.**

**Consistency**

**Data is in a consistent state when a transaction starts and when it ends.**

**For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.**

**Isolation**

**The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.**

**For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.**

**Durability**

**After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.**

**For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.**

Isolation Levels

**Dirty Read:**

**A Dirty read is a situation when a transaction reads data that has not yet been committed. For example, Let’s say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.**

**Non-Repeatable read:**

**Non-Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.**

**Phantom Read:**

**Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time**

**Simple example:**

* **User A runs the same query twice.**
* **In between, User B runs a transaction and commits.**
* **Non-repeatable read: The A row that user A has queried has a different value the second time.**
* **Phantom read: All the rows in the query have the same value before and after, *but different rows are being selected* (because B has deleted or inserted some).**

**Based on these phenomena, The SQL standard defines four isolation levels:**

* 1. **Read Uncommitted:**

**Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.**

* 1. **Read Committed:**

**This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.**

* 1. **Repeatable Read**

**This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.**

* 1. **Serializable**

**This is the highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.**

**Table

Description automatically generated**

Spring Caching

**In JPA and Hibernate, the first-level cache is a Java Map, in which the Map key represented by an object that encapsulates the entity name and its identifier, and the Map value is the entity object itself.**

**key = object that encapsulates the entity name and identifier**

**value = entity object itself**

**Hibernate is an open source object relational mapping (ORM) tool that provides a framework to map object-oriented domain models to relational databases for web applications.**

**First, Hibernate checks whether the entity is already stored in the first-level cache, and if it is, the currently managed entity reference is returned.**

**If the JPA entity is not found in the first level-cache, Hibernate will check the second-level cache if this cache is enabled.**

**If the entity is not found in the first or second-level cache, then Hibernate will load it from the database using an SQL query.**

**First Level Cache**

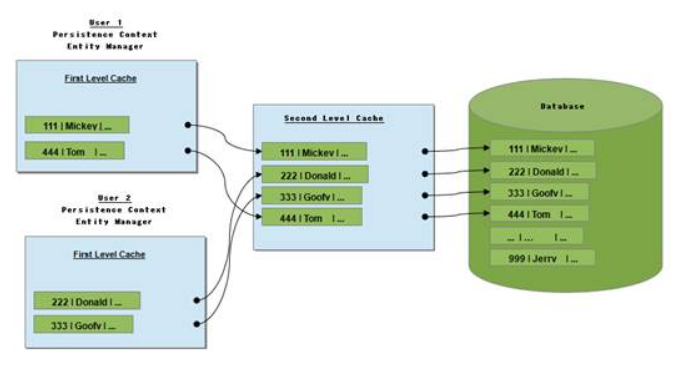
**Entity Manager always maintains a cache called the first level cache, so it makes sense to call another shared caching technique in addition to first – a second level cache. In first level cache CRUD operations are performed per transaction basis to reduce the number of queries fired to the database. That is, an entity modified several times within the same transaction is done in the cache only, modification at the database level is slated until final UPDATE statement is fired at the end of the transaction.**

**Diagram

Description automatically generated**

**Second level cache was created for performance more than out of necessity.**

**Second level cache sits between Entity Manager and the database. Persistence context shares the cache, making the second level cache available throughout the application. Database traffic is reduced considerably because entities are loaded into the shared cache and made available from there.**

****

Database Normalization

**First Form:**

**Graphical user interface, text, application, email

Description automatically generated**

**Second Form:**

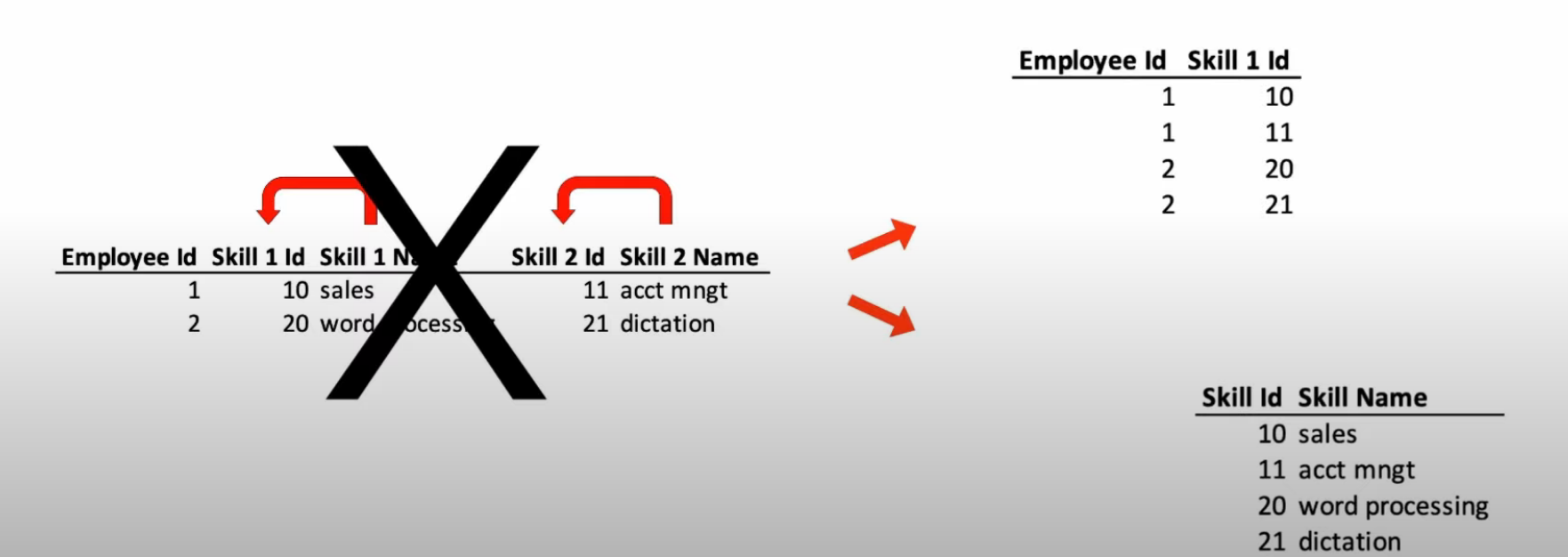
* **All data must depend on the Primary Key**
* **The second form eliminates Partial Dependency**

**A picture containing text

Description automatically generated**

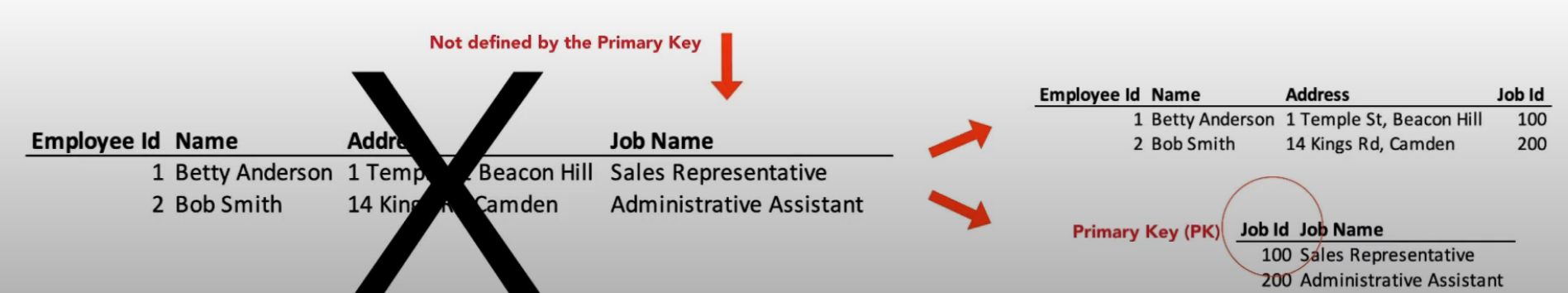
**Skill\_1\_name depends on Skill\_1\_Id, not on employee Id.**

* **Any column that doesn’t depend on the primary key must be split onto its own table.**



**Third Form:**

* **The primary key must fully define all non-key columns and non-key columns must not depend on any other key.**
* **The third form eliminates Transitive Dependency**

****

Spring Security Claims and Principal

[**https://www.baeldung.com/spring-security-method-security**](https://www.baeldung.com/spring-security-method-security)

**Authentication**:

**The process of identifying someone's identity by assuring that the person is the same as what he is claiming for.**

**Authorization:**

**The process of granting someone permission to do something. It means it a way to check if the user has permission to use a resource or not.**

**Principal:**

**The principal is the currently logged in user. However, you retrieve it through the security context which is bound to the current thread and as such it's also bound to the current request and its session.**

**@Secured**

**The @Secured annotation is used to specify a list of roles on a method.** **So, a user only can access that method if she has at least one of the specified roles.**

**Graphical user interface, text, application, email

Description automatically generated**

### **@RolesAllowed**

**Graphical user interface, text, application, email

Description automatically generated**

### **@PreAuthorize and PostAuthorize**

**The @PreAuthorize annotation checks the given expression before entering the method**, **whereas** **the @PostAuthorize annotation verifies it after the execution of the method and could alter the result.**