Transaction management and Caching.

What is transaction management?

Critically, transaction management software relies on the concept of atomicity to define a singular transaction as a set series of operations that must all be completed, or none are considered to be completed. In other words, to maintain data consistency, a transaction management system ensures that a transaction can never be partially completed.

For example, when a person attempts to withdraw money from an ATM, transaction management software processes the necessary database queries and changes in order to check their account balance, subtract the requested amount, update the bank's records and release the dispensed cash. All of these steps are considered one new transaction, and the transaction management system ensures that the entire process is completed to prevent any inconsistencies in the bank's database and preserve an accurate ledger

Transactions obey the following basic properties, known as ACID properties. ACID is an acronym for the following:

Atomicity

All tasks of a transaction are performed or none of them are. There are no partial transactions. For example, if a transaction starts updating 100 rows, but the system fails after 20 updates, then the database rolls back the changes to these 20 rows.

Consistency

The transaction takes the database from one consistent state to another consistent state. For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.

Isolation

The effect of a transaction is not visible to other transactions until the transaction is committed. For example, one user updating the hr.employees table does not see the uncommitted changes to employees made concurrently by another user. Thus, it appears to users as if transactions are executing serially.

Durability

Changes made by committed transactions are permanent. After a transaction completes, the database ensures through its recovery mechanisms that changes from the transaction are not lost.

The use of transactions is one of the most important ways that a database management system differs from a file system

States of transactions

While the transaction is in progress, and the database is in flux, the transaction can be broken down into a number of sequential transaction states.

Active

Once a transaction begins, it enters into an active state during which database read and write operations can occur.

Partially committed

Once all the necessary steps of a transaction are completed, the transaction is considered to be only partially committed until the master database is updated.

Committed

After a transaction is completed successfully, it is committed to the database, entering a committed state.

Failed

When a transaction fails to execute one or more of its operations or is aborted, it is considered to be in a failed state. A failed transaction will trigger a rollback, which undoes any database changes in progress.

Terminated

The final state for all transactions, a transaction in the terminated state is taken out of the system and can no longer perform any database operations.

Java Transaction API (JTA)

JTA is the standard application programming interface (API) for enterprise applications, allowing transaction management applications to communicate with other application types, including databases and messaging systems, while ensuring atomicity and consistency.

Uses of Transaction Management

- The DBMS is used to schedule the access of data concurrently. It means that the user can access multiple data from the database without being interfered with by each other. Transactions are used to manage concurrency.
- It is also used to satisfy ACID properties.
- It is used to solve Read/Write Conflicts.
- It is used to implement Recoverability, Serializability, and Cascading.
- Transaction Management is also used for Concurrency Control Protocols and the Locking of data.

Advantages of using a Transaction

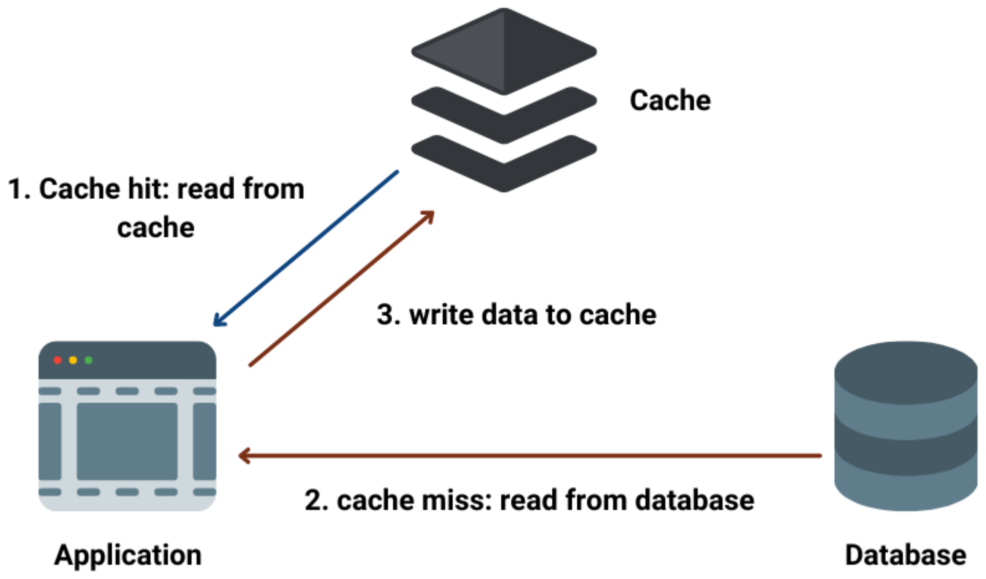
- Maintains a consistent and valid database after each transaction.
- Makes certain that updates to the database don't affect its dependability or accuracy.
- Enables simultaneous use of numerous users without sacrificing data consistency.

Disadvantages of using a Transaction

- It may be difficult to change the information within the transaction database by end-users.
- We need to always roll back and start from the beginning rather than continue from the previous state.

What is caching?

Caching is when a software component stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. **A cache hit** occurs when the requested data can be found in a cache, while a **cache miss** occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests that can be served from the cache, the faster the system performs


Cache
1. Cache hit: read from cache
3. write data to cache
Application
2. cache miss: read from database
Database

Caching

Spring Boot Caching

Spring Framework provides caching in a Spring Application, transparently. In Spring, the **cache abstraction** is a mechanism that allows consistent use of various caching methods with minimal impact on the code.

Cache Abstraction

The cache abstraction mechanism applies to Java **methods**. The main objective of using cache abstraction is to **reduce** the number of executions based on the information present in the cache. It applies to expensive methods such as **CPU** or **IO bound.**

Every time, when a method invokes, the abstraction applies a cache behavior to the method. It checks whether the method has already been executed for the given argument or not.

What data should be cached?

- The data that do not change frequently.

- The frequently used read query in which results does not change in each call, at least for a period.

Types of Caching

There are **four** types of caching are as follows:

- In-memory Caching
- Database Caching
- Web server Caching
- CDN Caching

@Cacheable

It is a method level annotation. It defines a cache for a method's return value. The Spring Framework manages the requests and responses of the method to the cache that is specified in the annotation attribute. The @Cacheable annotation contains more options. For example, we can provide a **cache name** by using the **value** or **cacheNames** attribute.

We can also specify the **key** attribute of the annotation that uniquely identifies each entry in the cache. If we do not specify the key, Spring uses the default mechanism to create the key.

Example

In the following example, we have cached the **return value** of the method **studentInfo()** in **cacheStudentInfo,** and **id** is the unique key that identifies each entry in the cache.

```
@Cacheable(value="cacheStudentInfo", key="#id")
public List studentInfo()
{
return studentDetails;
}
```

We can also apply a condition in the annotation by using the condition attribute. When we apply the condition in the annotation, it is called **conditional caching**.

For example, the following method will be cached if the argument name has a length shorter than 20.

```
@Cacheable(value="student", condition="#name.length<20")
public Student findStudent(String name)
{
//some code
}
```

@CacheEvict

It is a method level annotation. It is used when we want to remove stale or unused data from the cache. It requires one or multiple caches that are affected by the action. We can also specify a key or condition into it. If we want wide cache eviction, the @CacheEvict annotation provides a parameter called **allEntries**. It evicts all entries rather than one entry based on the key.

One important point about @CacheEvict annotation is that it can be used with void methods because the method acts as a trigger. It avoids return values. On the other hand, the annotation @Cacheable requires a return value that adds/updates data in the cache. We can use @CacheEvict annotation in the following ways:

Evict the whole cache:

```
@CacheEvict(allEntries=true)
```

Evict an entry by key:

```
@CacheEvict(key="#student.stud_name")
```

Example

The following annotated method evicts all the data from the cache **student_data**.

```
@CacheEvict(value="student_data", allEntries=true) //removing all entries from the cache
public String getNames(Student student)
{
//some code
}
```

@CachePut

It is a method level annotation. It is used when we want to **update** the cache without interfering the method execution. It means the method will always execute, and its result will be placed into the cache. It supports the attributes of @Cacheable annotation.

A point to be noticed that the annotations @Cacheable and @CachePut are not the same because they have different behavior. There is a slight difference between @Cacheable and @CachePut annotation is that the @**Cacheable** annotation **skips the method execution** while the **@CachePut** annotation **runs the method** and put the result into the cache.

Example

The following method will update the cache itself.

```
@CachePut(cacheNames="employee", key="#id") //updating cache
public Employee updateEmp(ID id, EmployeeData data)
{
//some code
}
```

References

1. Wikipedia contributors. (n.d.). *Cache (computing)*. In *Wikipedia, The Free Encyclopedia*. Retrieved September 4, 2024, from https://en.wikipedia.org/wiki/Cache_(computing)

2. Javatpoint. (n.d.). *Spring Boot caching*. Retrieved September 4, 2024, from https://www.javatpoint.com/spring-boot-caching

3. Oracle Corporation. (n.d.). *Introduction to transactions*. In *Oracle® Database Concepts 21c*. Retrieved September 4, 2024, from https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/transactions.html#GUID-A049FE81-8B67-4386-B599-9CDD7E6B6C59

4. IBM. (n.d.). *Transaction management*. Retrieved September 4, 2024, from https://www.ibm.com/topics/transaction-management

5. GeeksforGeeks. (n.d.). *Transaction management*. Retrieved September 4, 2024, from https://www.geeksforgeeks.org/transaction-management/