Caching Mechanism in Hibernate

What is Hibernate

- An ORM tool which provides a framework for mapping an object-oriented domain model to a relational database
- Simplifies the data creation, data manipulation and data access
- Reduces the length of code with increased readability
- Generates database independent queries
- High performance due to multi layered cache usage

Caching

- The process of storing copies of data in a temporary storage (cache) for quicker access
- Lies between the application and database
- Significantly smaller than the main memory or database
- Relatively faster and more expensive than the actual data store
- Cost efficiency is a trade-off between disk size and speed

Caching in Hibernate: At a Glance

- A temporary memory that keeps a representation of current database state close to the application
- Data is stored in a hydrated (disassembled) state
- To Hibernate a cache looks like a map of key/value pairs
- Store data in the cache by providing a key and a value
- Lookup a value in the cache with a key
- Utilizes multi level caching
    - First level cache (default)
    - Second level cache (optional)

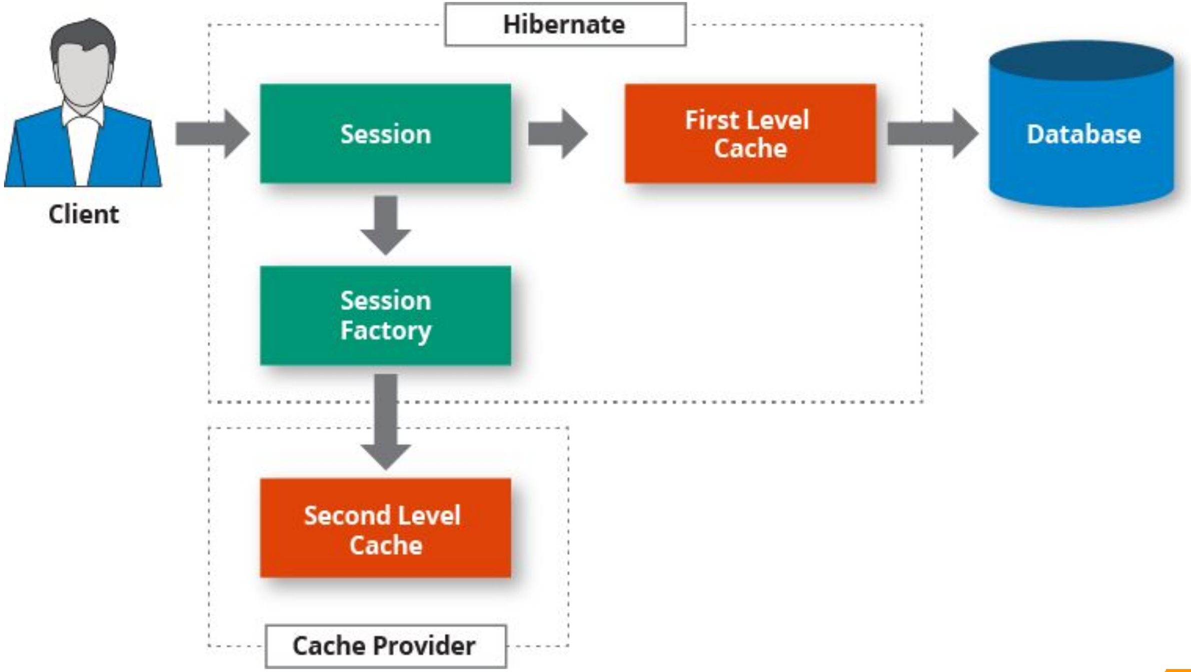
Caching in Hibernate: First Level Cache

- Enabled by default
- Associates with the Session object
- Any object cached in a session will not be visible to other sessions
- When the session is closed, all the cached objects will also be lost
- An application can use many session object
- No DB hit when we're getting same data in same session
- Might get a DB hit when getting same data in different session

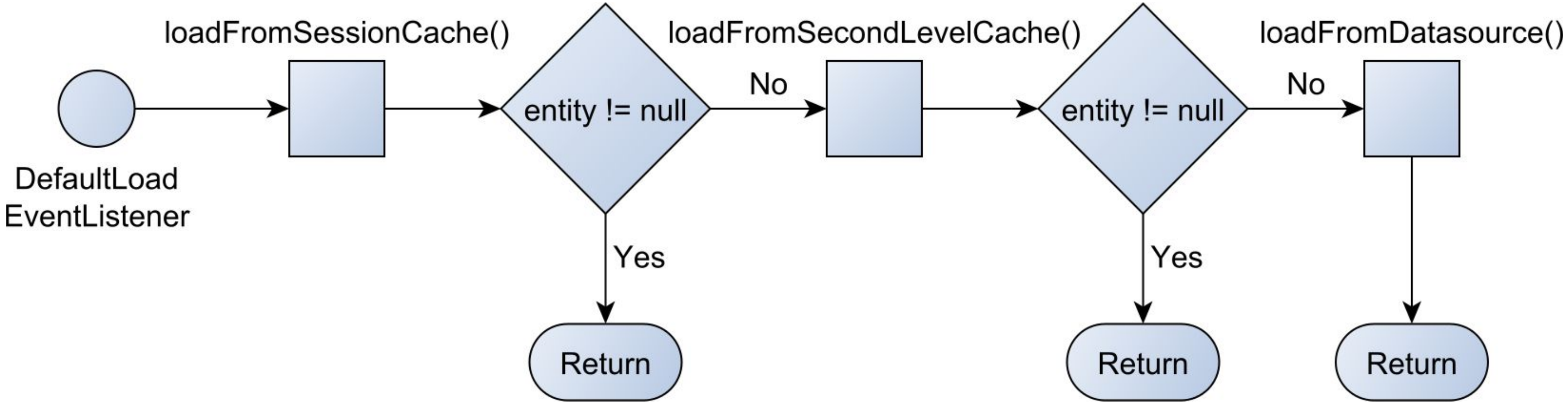
Caching in Hibernate: Second Level Cache

- Not enabled by default
- Associates with the Session Factory object
- SessionFactory is global for all session objects
- Uses a common cache for all the session objects of a session factory
- Stores data in the cache as a map of key/value pairs
- Can be configured on a per-class and per-collection basis
- Can choose among multiple third party cache providers

Caching in Hibernate: Representation



Caching in Hibernate: **Flow Chart**



The Second Level Cache

Types, Strategies and Configuration of Hibernate’s Second Level Cache

Types of Second Level Cache

- **Entity Data Cache:** The application performs an entity instance lookup by identifier (Primary Key)
  - Cache key is the identifier value
  - Cache Value is Entity’s property values
  - Stores cached entities in a dehydrated (disassembled) format
  - Assembles the instance when it reads from the cache
- **Collection Cache:** When a collection is initialized
  - Cache key is the Collection role (i.e.: IndividualPlan[123]#changeForms)
  - Cache value is a set of identifier values of the collection (set of Change Form Ids)
- **Natural Identifier Cache:** When the application performs an entity instance lookup by a unique key attribute
  - Cache key is the unique property (i.e: Form ID)
  - Cached value is the entity instance identifier

Types of Second Level Cache

- **Query Result Cache:**
  - When a JPQL, Criteria or SQL query is executed
  - Cache key is the query with its params
  - Cache value is the query result set (may include identifier values)
  - It’s disabled by default.
  - Caches only identifiers, not the actual entity state.
  - Should be used in association with other second level cache

Cache Concurrency Strategies (1/4)

- **READ_WRITE**
  - Caching will work for both read and write
  - A soft lock is stored in the cache for the entity which is being updated, and released after the commit
  - Concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database
  - Maintains read committed isolation
  - Used for read-mostly data (where it’s critical to prevent stale data in concurrent transactions)
  - Shouldn’t enable this strategy if data is concurrently modified (by other applications) in the database

Cache Concurrency Strategies (2/4)

- **NONSTRICT_READ_WRITE**
  - Used when its extremely unlikely that two transactions would try to update the same item simultaneously
  - Cache is updated after a transaction has been committed
  - No guarantee of consistency between the cache and the database
  - A transaction may read stale data from the cache
  - Can configure the duration of the inconsistency window with the expiration policies of your cache provider

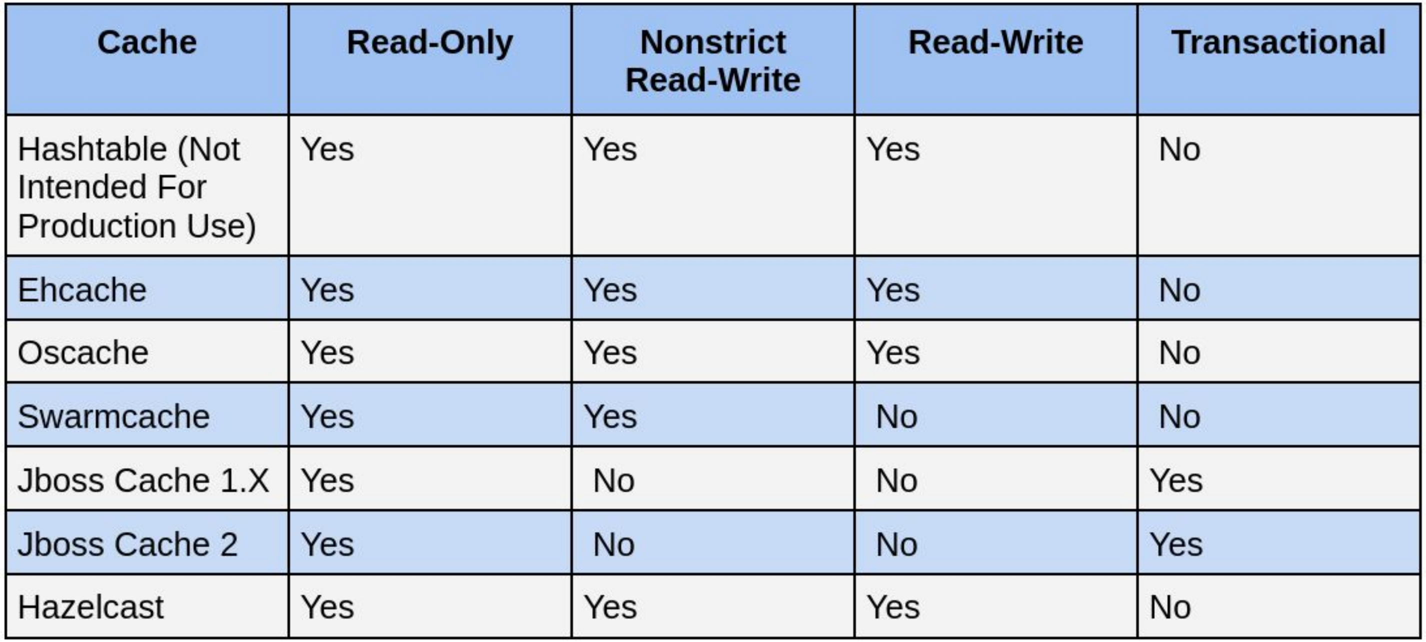
Cache Concurrency Strategies (3/4)

- **READ_ONLY**
  - Very suitable for some static reference data that don't change
  - Throws an exception if an update is triggered
  - Simplest and best performing strategy
  - Perfectly safe for use in a cluster

Cache Concurrency Strategies (4/4)

- **TRANSACTIONAL**
  - Available only in environments with a system transaction manager
  - Guarantees full transactional isolation up to repeatable read
  - Hibernate does not provide any locking or version checking
  - Assumes cache provider manages all the transactions
  - Used for read-mostly data

Cache-Provider/Concurrency-Strategy Compatibility



Cache	Read-Only	Nonstrict Read-Write	Read-Write	Transactional
Hashtable (Not Intended For Production Use)	Yes	Yes	Yes	No
Ehcache	Yes	Yes	Yes	No
Oscache	Yes	Yes	Yes	No
Swarmcache	Yes	Yes	No	No
Jboss Cache 1.X	Yes	No	No	Yes
Jboss Cache 2	Yes	No	No	Yes
Hazelcast	Yes	Yes	Yes	No

Hibernate Cache Best Practices

Caching is useful when:

- DB is updated only via Hibernate (Cache is not invalidated if the database is updated in a different way)
- Data is updated rarely but it’s read frequently
- Noncritical data where slight inconsistency is allowed

Caching is not recommended for:

- Critical data that is updated often
- Financial data, where decisions must be based on the latest update
- Queries with lots of different combinations of parameter values

Evicting/Clearing Caches Manually

- sessionFactory.getCache().containsEntity(Cat.class, catId): Is this particular Cat currently in the cache
- sessionFactory.getCache().evictEntity(Cat.class, catId): Evict a particular Cat
- sessionFactory.getCache().evictEntityRegion(Cat.class): Evict all Cats
- sessionFactory.getCache().evictEntityRegions(): Evict all entity data
- sessionFactory.getCache().containsCollection("Cat.kittens", catId): Is this particular collection currently in the cache
- sessionFactory.getCache().evictCollection("Cat.kittens", catId): Evict a particular collection of kittens
- sessionFactory.getCache().evictCollectionRegion("Cat.kittens"): Evict all kitten collections
- sessionFactory.getCache().evictCollectionRegions(): Evict all collection data

Configuring L2 Cache in Hibernate (1/4)

Enabling second level cache and cache provider in *persistence.xml*

```
<properties>
    ...
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="hibernate.cache.region.factory_class"
      value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
    ...
</properties>
```

Making an Entity Cacheable

```
@Entity
@Table(name = "client")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE, region = "hibernate-very-big-client")
public class Client {
...
```

Configuring L2 Cache in Hibernate (2/4)

Custom configuration (based on cache names) in *hazelcast-hibernate.xml*

```
<map name="hibernate-very-big-*">
        <backup-count>0</backup-count>
        <time-to-live-seconds>1800</time-to-live-seconds>
        <max-idle-seconds>600</max-idle-seconds>
        <eviction-policy>LRU</eviction-policy>
        <max-size policy="PER_NODE">3000</max-size>
        <near-cache>
            <time-to-live-seconds>600</time-to-live-seconds>
            <max-idle-seconds>300</max-idle-seconds>
            <eviction size="1500" eviction-policy="LRU"/>
            <invalidate-on-change>true</invalidate-on-change>
        </near-cache>
    </map>
```

Configuring L2 Cache in Hibernate (3/4)

Custom configuration parameters

- **Backup Count** (0, 1, 2): Number of replicated instances for each data
- **Time To Live**: Maximum time in seconds for each entry to stay in the map
- **Max Idle Seconds**: Maximum time in seconds for each entry to stay idle in the map
- **Eviction Policy**
  - NONE (Default): No items are evicted and the property Max Size described is ignored
  - LRU: Least Recently Used.
  - LFU: Least Frequently Used
- **Max Size:** Max number of key-value pairs stored in the cache

Configuring L2 Cache in Hibernate (4/4)

Custom configuration parameters

- **Near Cache**
  - Cache entries in Hazelcast are partitioned across the cluster members
  - Retrieving is costly (remote operation) if a key-value pair is owned by another member in your cluster
  - For mostly-read data, creating local “Near Caches” might be efficient
  - Increases retrieval speed at the cost of higher memory consumption
- **Invalidate on Change:** Whether the cached entries are evicted when the entries are updated or removed

😁

THANKS!