Second Level Cache

https://vladmihalcea.com/how-does-hibernate-store-second-level-cache-entries/

https://www.baeldung.com/java-object-hydration

https://www.baeldung.com/spring-cache-tutorial

https://docs.jboss.org/hibernate/orm/4.3/javadocs/org/hibernate/annotations/CacheConcurrencyStrategy.html

Concurrency strategy is used in @Cacheable

The article explores how Hibernate stores second-level cache entries internally and provides insights into its implementation details. Here's a summary of the key points discussed in the article:

1. **Introduction to Second-Level Cache**:
   - Hibernate provides a second-level cache mechanism to improve performance by caching entities and query results at a session factory level.
   - Second-level cache is shared across different sessions and transactions, allowing cached data to be reused by multiple users.

2. **Caching Entity Data**:
   - Hibernate stores entity data in the second-level cache using a map-like structure where the cache key is composed of the entity type (class) and the entity identifier (primary key).
   - When an entity is retrieved from the database, Hibernate checks if it exists in the second-level cache by generating a cache key based on the entity type and identifier.
   - If the entity is found in the cache, it is returned directly without hitting the database, improving performance.

3. **Caching Query Results**:
   - Hibernate also supports caching query results in the second-level cache.
   - When a query is executed, Hibernate generates a cache key based on the query string, query parameters, and other factors.
   - If the query result set is found in the cache, it is returned directly, bypassing the database query execution.

4. **Cache Region Management**:
   - Hibernate organizes second-level cache entries into cache regions, each corresponding to a specific entity type or query.
   - Cache regions are managed by a cache region factory, which is responsible for creating and managing cache regions.

5. **Cache Entry Structure**:
   - Each cache entry in the second-level cache consists of metadata and the cached data itself.
   - The metadata includes information such as the timestamp of the last cache update, the expiration time, and the cache entry status.

6. **Cache Entry Lifecycle**:
   - Hibernate manages the lifecycle of cache entries, including creation, update, eviction, and invalidation.
   - Cache entries are automatically evicted or invalidated based on configurable policies such as time-to-live (TTL) or maximum entry count.

7. **Concurrency Considerations**:
   - Hibernate handles concurrency issues in the second-level cache by using synchronization mechanisms to ensure thread safety.
   - Concurrent access to cache entries is managed to prevent race conditions and maintain data consistency.

8. **Configuration Options**:
   - Hibernate provides various configuration options for customizing the behavior of the second-level cache, including cache providers, eviction policies, and cache region settings.

Overall, the article provides a comprehensive overview of how Hibernate implements and manages the second-level cache, highlighting its benefits and considerations for effective caching strategies in Hibernate-based applications.

Summary of the link:

In Hibernate's second-level cache, each entity is stored as a `CacheEntry`. The entity's hydrated state, which represents its data at the time of retrieval from the database, is used to create the cache entry value. Let's break down the process:

1. **Hydration**:
   - Hydration refers to the transformation of a JDBC ResultSet into an array of raw values.
   - When an entity is loaded from the database, Hibernate hydrates its state using a JDBC ResultSet.
   - The hydrated state is saved in the currently running Persistence Context as an `EntityEntry` object, which encapsulates the loading-time entity snapshot.

2. **Usage of Hydrated State**:
   - The hydrated state is used by Hibernate for:

- Default dirty checking mechanism: It compares the current entity data against the loading-time snapshot to detect changes.
- Second-level cache: Cache entries are built from the loading-time entity snapshot, allowing efficient caching and retrieval of entities.

3. **Dehydration**:
   - The inverse operation of hydration is called dehydration.
   - Dehydration involves copying the entity state from memory into an INSERT or UPDATE statement when persisting changes back to the database.

4. **Second-level Cache Elements**:
   - Hibernate's second-level cache elements use a disassembled hydrated state instead of full entity graphs.
   - The disassembled state is obtained from the hydrated state prior to being stored in the cache entry.
   - Disassembling involves breaking down the hydrated state into its constituent parts, such as individual property values and associations.

5. **Example**:
   - Suppose we have entities like `Post`, `PostDetails`, and `Comment`.
   - When we persist a `Post` entity with associated `PostDetails` and `Comment` entities, each of these entities will have its hydrated state stored in the second-level cache as a `CacheEntry`.
   - The hydrated state of each entity contains its data at the time of retrieval, which can be efficiently stored and retrieved from the cache.

In summary, Hibernate's second-level cache stores entities as `CacheEntry` objects, using their hydrated state obtained during retrieval from the database. This allows for efficient caching and retrieval of entity data, improving application performance by reducing database access.

The *Post* entity cache element Explanation

In the example provided, the cache element for the `Post` entity is shown after fetching a `Post` entity from the database. Let's dissect the cache element and its components:

1. **Cache Key**:
   - The cache key is represented by an instance of `org.hibernate.cache.spi.CacheKey`.
   - It contains metadata about the cached entity, including its type (`entityOrRoleName`) and identifier (`1` in this case).

2. **Cache Entry**:
   - The cache entry is represented by an instance of `org.hibernate.cache.spi.entry.StandardCacheEntryImpl`.
   - It contains the disassembled hydrated state of the `Post` entity, which

includes the values of its properties (`name` column and `id` in this case).
   - The disassembled state is an array of Serializable objects representing the entity's properties. In this case, it includes the identifier (`1`) and the name (`"Hibernate Master Class"`).

3. **Associations**:
   - The cache entry does not embed the associations of the `Post` entity, such as the one-to-many association with `Comment` entities or the inverse one-to-one association with `PostDetails`.
   - The cache entry only contains the direct properties of the `Post` entity (`id` and `name`), which are necessary for reconstructing the entity's state upon cache retrieval.

4. **Lazy Properties**:
   - The `lazyPropertiesAreUnfetched` attribute indicates whether any lazy properties of the entity are unfetched. In this case, it is set to `false`, suggesting that all properties are eagerly fetched.

5. **Version**:
   - The `version` attribute is null, indicating that the entity does not have a version property.

In summary, the cache element for the `Post` entity contains the disassembled hydrated state of the entity, which includes its identifier and properties. Associations and lazy properties are not embedded in the cache entry, as they are not part of the entity's direct state and are managed separately by Hibernate. This cache element allows for efficient caching and retrieval of `Post` entities from the second-level cache.