# **Spring Data II Report**

**—----------------------------------------------------------------------------------------**

Each of these fetching strategies has its own trade-offs in terms of the number of SQL queries, the amount of data transferred from the database, and the performance of the operation. The best strategy to use depends on the specific requirements of your application and the characteristics of your data.

### SELECT

* 1. **Implementation:** The getBySelectBenchmark method in the ProductService class fetches all ProductSelect entities from the database. The entity uses @Fetch(FetchMode.SELECT) to retrieve Reviews. The Hibernate query generated for this operation is a simple SELECT statement to fetch all rows from the ProductSelect table.
  2. **Observation:** It found 1000 products and for each product, it also fetched 100 related reviews. This resulted in a total of 1000 SQL queries, which is exactly the same number as the total number of products fetched along with their associated reviews.
  3. **Performance:** The select operation took 1585 milliseconds to complete. During the operation, approximately 158476 megabytes of memory were utilized. A total of 1000 SQL queries were executed during the select operation.
  4. **Practical Use:** This select method would be ideal for scenarios where you need to retrieve a large set of data from a database, including associated entities, and analyze their properties or relationships.

### JOIN

* 1. **Implementation:** The getByJoinBenchmark method in the ProductService class fetches all ProductJoin entities from the database. The entity uses @Fetch(FetchMode.JOIN) to retrieve Reviews. The Hibernate query generated for this operation is a simple SELECT statement to fetch all rows from the ProductJoin table.
  2. **Observation:** It found 1000 products and for each product, it also fetched 100 related reviews. This resulted in a total of 1000 SQL queries, which is exactly the same number as the total number of products fetched along with their associated reviews.
  3. **Performance:** The join operation took 1750 milliseconds to complete. During the operation, approximately 197616 megabytes of memory were utilized. A total of 1000 SQL queries were executed during the join operation.
  4. **Practical Use:** This join method would be well-suited for scenarios where you need to retrieve data from multiple related tables or entities in a database and combine them into a single result set.

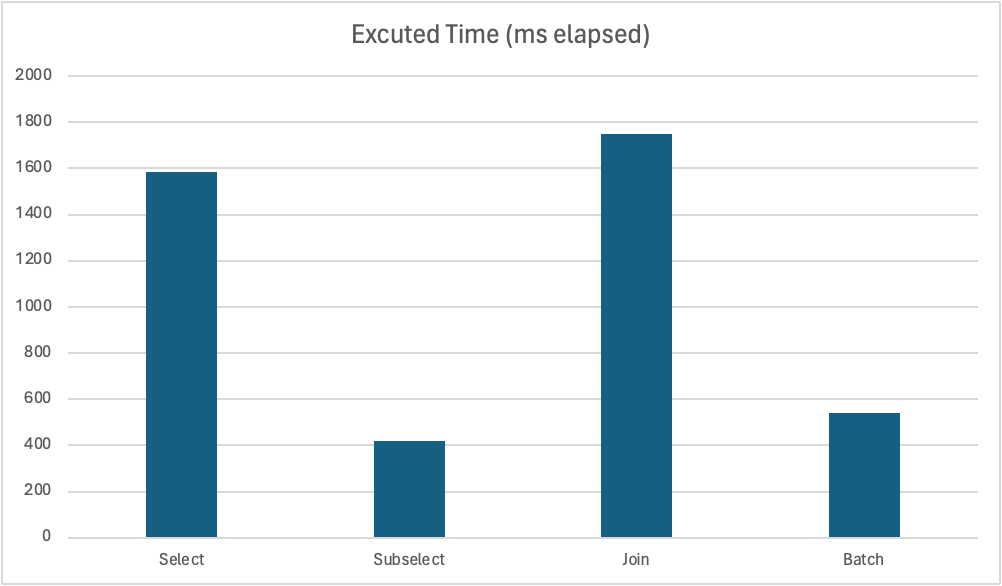
### SUBSELECT

* 1. **Implementation:** The getBySubselectBenchmark method in the ProductService class fetches all ProductSubselect entities from the database. The entity uses @Fetch(FetchMode.SUBSELECT) to retrieve Reviews. The Hibernate query generated for this operation is a simple SELECT statement to fetch all rows from the ProductSubselect table.
  2. **Observation:** This approach efficiently retrieved data for 1000 products and their associated 100 reviews each. Impressively, only 2 SQL queries were executed during the entire operation, showcasing the effectiveness of the subselect fetch strategy in minimizing database interactions and optimizing performance.
  3. **Performance:** The subselect method completed in just 421 milliseconds. During the operation, approximately 217727 megabytes of memory were utilized. Only 2 SQL queries were executed, demonstrating the efficiency of the subselect method in minimizing database interactions.
  4. **Practical Use:** The subselect method is particularly useful in scenarios where you need to retrieve data from a database along with its associated entities, but the associated entities are rarely accessed or the number of associated entities varies greatly.

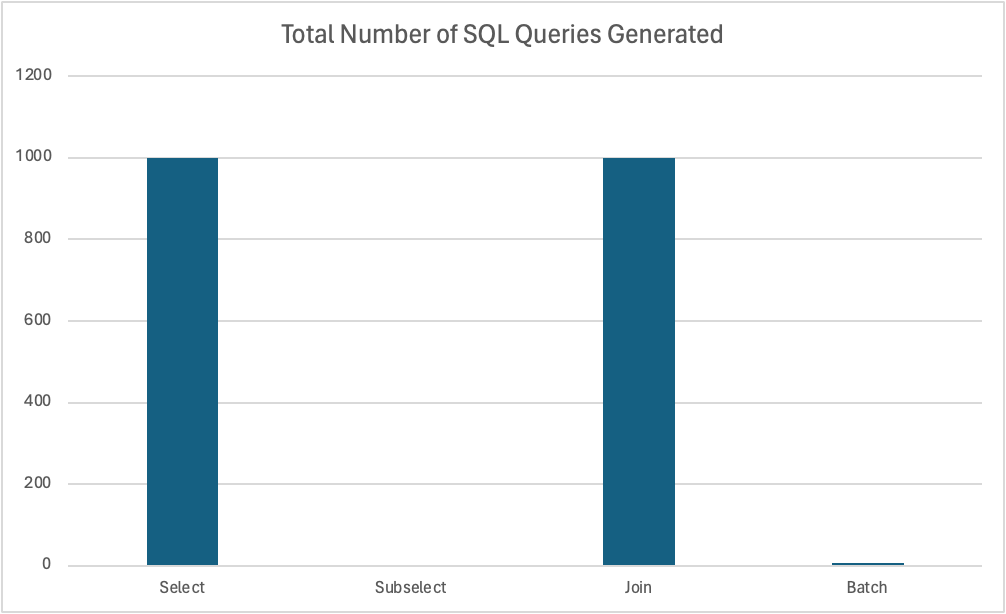
### BATCH

* 1. **Implementation:** The getByBatchBenchmark method in the ProductService class fetches all ProductBatch entities from the database. The entity uses @BatchSize(size = 200) to retrieve Reviews. The Hibernate query generated for this operation is a simple SELECT statement to fetch all rows from the ProductBatch table.
  2. **Observation:** This approach effectively retrieved data for 1000 products, each with 100 associated reviews. Notably, the batch fetch strategy resulted in 6 SQL queries being executed throughout the operation, suggesting a moderate level of database interaction.
  3. **Performance:** The batch method completed in 540 milliseconds, showing reasonable speed in retrieving the data. During the operation, approximately 134214 megabytes of memory were used, indicating a moderate level of resource consumption. A total of 6 SQL queries were executed, which suggests a moderate level of interaction with the database.
  4. **Practical Use:** The batch method is well-suited for scenarios where you need to retrieve a large amount of data from a database, including associated entities, with moderate efficiency. Especially in scenarios where the number of associated entities varies.

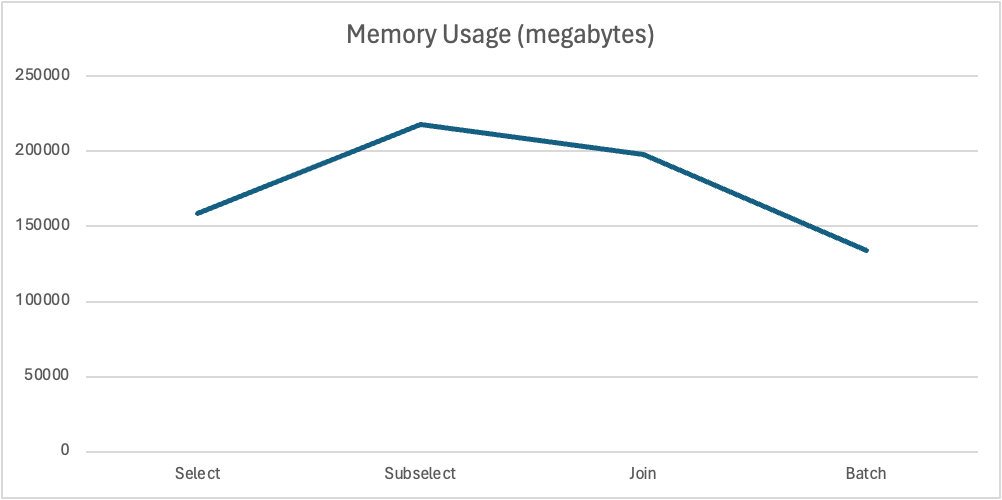
### Query Execution Time Comparison



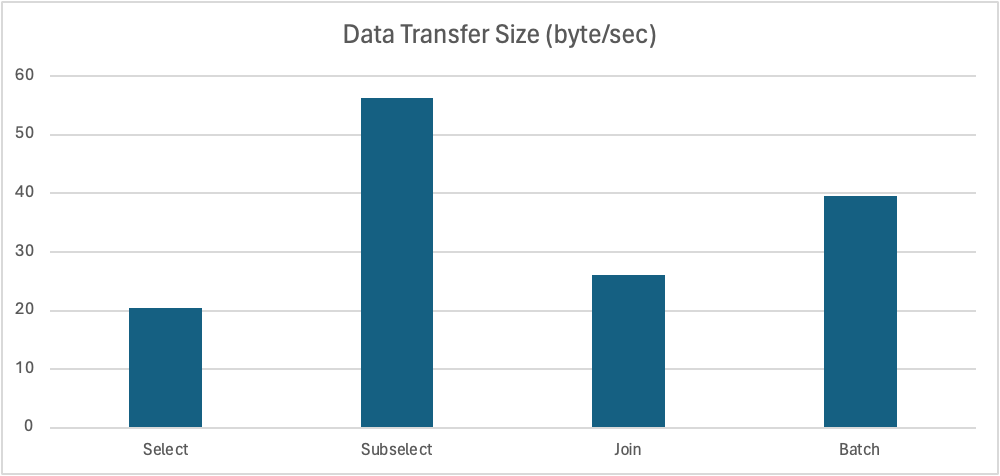
### Total Number of SQL Queries Generated



### Memory Usage



### Data transfer size



### Summary

The Spring Data JPA II Report includes a comparison of four different fetching strategies for retrieving data from a database: getBySelect(), getByJoin(), getBySubselect(), and getByBatch(). Each strategy has its own trade-offs in terms of the number of SQL queries, the amount of data transferred from the database, and the performance of the operation. The report provides detailed observations, performance metrics, and practical use cases for each strategy, allowing developers to make informed choices based on their specific requirements.