In Apache Spark, there are several sorting strategies available for joining datasets. The main goal is to minimise data shuffling and improve parallelism performance. Some of the common sorting strategies in Spark include:

**Sort Merge Join:** The Sort Merge Join strategy in Spark is used to join two datasets by sorting both datasets on the join keys and then merging them together. By partitioning and sorting the datasets, the amount of data that needs to be shuffled between executors is minimised during the operation, resulting in better performance compared to other join strategies like the Shuffle Hash Join. However, sorting can be expensive, especially for large datasets, so the Sort Merge Join may incur higher computational costs during the sorting phase.

1. **Partitioning**: Both datasets are partitioned based on the join keys. Spark ensures that the records with the same join key are present in the same partition across both datasets. This ensures that records with the same join key are co-located within the same executor.
2. **Sorting**: Within each partition, both datasets are sorted based on the join keys. This sorting step is crucial for the merge phase to efficiently combine the records.
3. **Merge Phase**: Once the datasets are sorted, Spark performs a merge operation. It iterates through both sorted datasets simultaneously, comparing the join keys and upon finding matching join keys, it combines the corresponding records into the result set.
4. **Output**: The result of the operation is a new dataset that combines the matching records from both datasets based on the join keys.

**Broadcast Hash Join:** It is used for joining large and small datasets efficiently. It works by broadcasting the smaller dataset to all the nodes in the cluster and then performing a join operation with the larger dataset locally on each node. One of the datasets should be smaller and can fit entirely into the memory of each node in the cluster, while the other dataset is too large to fit in memory and needs to be partitioned across the cluster. It reduces the overhead of shuffling large amounts of data across the network but if the smaller dataset is too large to fit into memory, broadcasting it can lead to out-of-memory errors.

1. **Broadcasting**: Spark determines which dataset is smaller and can fit into memory. This smaller dataset is then broadcasted to all the nodes in the cluster where a copy of the smaller dataset is sent to each executor's memory.
2. **Partitioning**: The larger dataset is partitioned across the cluster based on the join key. Each partition of the larger dataset is then processed on the corresponding executor.
3. **Local Join**: On each executor, the partition of the larger dataset is joined with the entire broadcasted smaller dataset locally. Since the smaller dataset is available in

memory on each executor, the join operation can be performed efficiently without the need for shuffling data across the network.

4. **Output**: The result of the operation is a new dataset that combines the matching records from both datasets based on the join keys.

**Shuffle Hash Join:** It is used for joining two large datasets efficiently by leveraging hash partitioning and shuffling to distribute and join the datasets across the cluster. It can incur performance overhead due to the shuffling process, especially if the network bandwidth is limited or if the data skew is significant.

1. **Partitioning**: Spark partitions both datasets based on the join keys. Each partition contains records with the same join key.
2. **Hashing**: Spark computes a hash value for each record in both datasets based on the join keys which determines which partition each record belongs to. Records with the same join key hash to the same partition.
3. **Shuffling**: Spark shuffles the partitions across the cluster, ensuring that records with the same join key are co-located on the same executor. This involves moving data between nodes in the cluster, which can incur network overhead.
4. **Local Join**: Once the data is shuffled and co-located, each executor performs a local join operation on the corresponding partitions of both datasets. This local join combines the matching records based on the join keys within each partition.
5. **Output**: The result of the local join operations is a set of partially joined partitions. Spark then performs a final shuffle to bring together all the matching records across partitions, ensuring that each record is joined with its corresponding records from the other dataset.
6. **Final Merge**: The partially joined partitions are merged to combined records from both datasets based on the join keys.

**Broadcast Nested Loop Join:** It is used for joining a large dataset with a small dataset. It is similar to the Broadcast Hash Join except for how they handle the join operation after broadcasting the smaller dataset. The Nested Loop Join can be less efficient compared to other join strategies, especially if the broadcasted dataset is significantly larger or if the join keys are not well-distributed.

1. **Broadcasting**: Spark determines which dataset is smaller and can fit into memory. The smaller dataset is then broadcasted to all the nodes in the cluster. Broadcasting involves sending a copy of the smaller dataset to each executor's memory.
2. **Partitioning**: The larger dataset is partitioned across the cluster based on the join key. Each partition of the larger dataset is processed on the corresponding executor.

3. **Nested Loop Join**: On each executor, the partition of the larger dataset is joined with the entire broadcasted smaller dataset using a nested loop join. This means that for each record in the larger dataset partition, Spark iterates over all records in the broadcasted smaller dataset to find matching records based on the join keys.
4. **Output**: The result of the join operation is a new dataset that combines the matching records from both datasets based on the join keys.

Considering the size of Dataset A (~1 million rows) and the potential for data skew, a Shuffle Hash Join might be a suitable choice. It can handle data skew by redistributing data across partitions and leveraging hash partitioning for efficient join operations. However, if the data skew is limited to only one geographical location within Dataset A, Broadcast Hash Join would also be a good choice as it is likely be more efficient given that Dataset B is relatively small. It would be essential to monitor the performance and resource utilization during both join operation to ensure efficient execution, especially considering the large size of Dataset A.