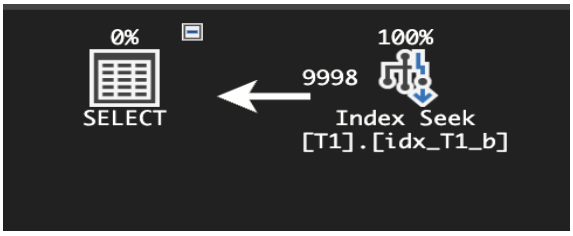
Project Part 3

Query 1:

```
SELECT T1.a, T1.b
FROM T1
WHERE T1.b < 10000
GO
```

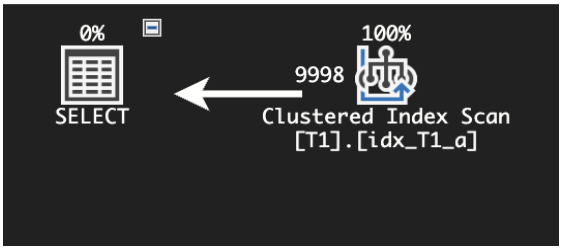
Query Plan 1:



Query 1 with hints:

```
SELECT T1.a, T1.b
FROM T1 WITH (INDEX(0))
WHERE T1.b < 10000
GO
```

Optimized Query Plan 1:



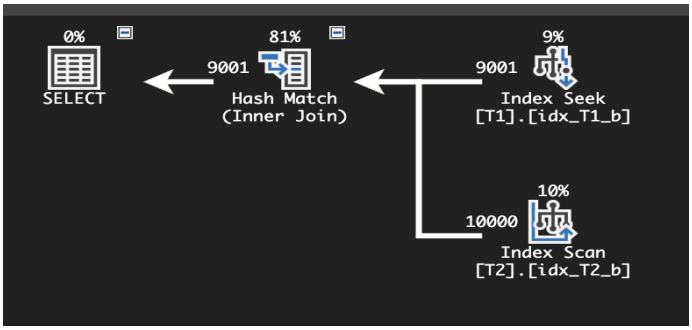
The reason that the second query may run faster than the first one can be attributed to the fact that where T1.b < 10000 will most likely include the majority of the records in the table. Thus, seeking and reverse scanning makes less sense than scanning the entire table from top to bottom. Now even though the Optimizer maintains stats on each attribute it decided to do an index seek which doesn't seem optimal to me thus I enforced a table Scan using hint INDEX(0).

Query 2:

```
SELECT T1.a, T2.b
```

```
FROM T1
JOIN T2 ON T1.a = T2.a
WHERE T1.b > 1000
GO
```

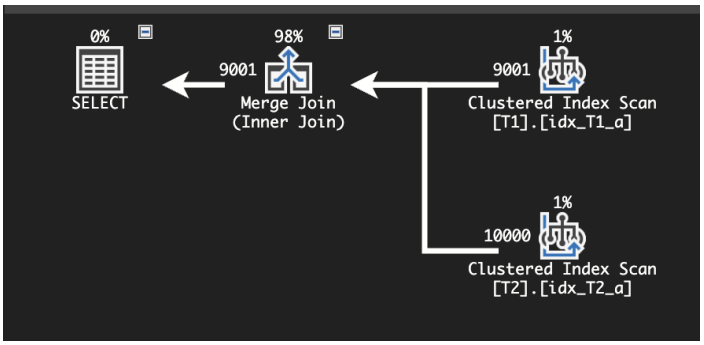
Query Plan 2:



Query 2 with hints:

```
SELECT T1.a, T2.b
FROM T1
JOIN T2 ON T1.a = T2.a
WHERE T1.b > 1000
OPTION (MERGE JOIN)
GO
```

Optimized Query Plan 2:



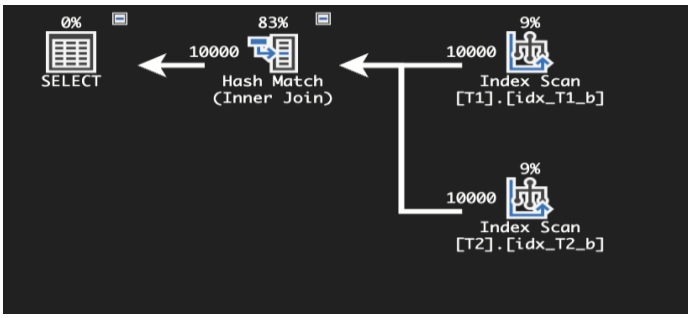
The Second query may run faster in this instance because it utilizes the existing indexes on the attributes and then merges them into the output table. As they are already sorted it will not require any additional sorting and will save time. The first query utilizes, hash join which not only requires additional time for hashing and storing the tables but also utilizes additional storage for storing the hash values which might not be useful in this case as the tables are only 10000 records long but may be a viable option once the cardinality of the tables increases.

Query 3:

```
SELECT T1.a, T2.b
FROM T1
```

```
INNER JOIN T2 ON T1.a = T2.a
WHERE T1.b > 500 OR T2.b < 3000
GO
```

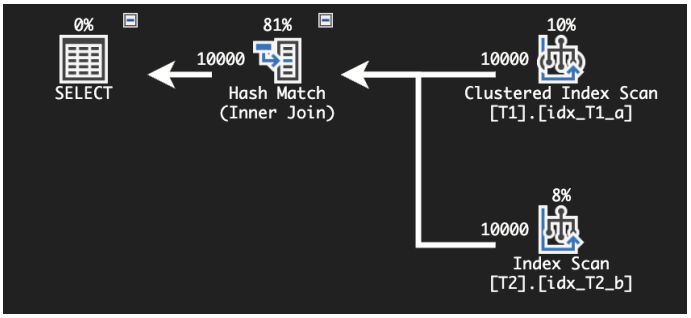
Query Plan 3:



Query 3 with hints:

```
SELECT T1.a, T2.b
FROM T1 WITH (INDEX(0))
INNER JOIN T2 ON T1.a = T2.a
WHERE T1.b > 500 OR T2.b < 3000
GO
```

Optimized Query Plan 3:



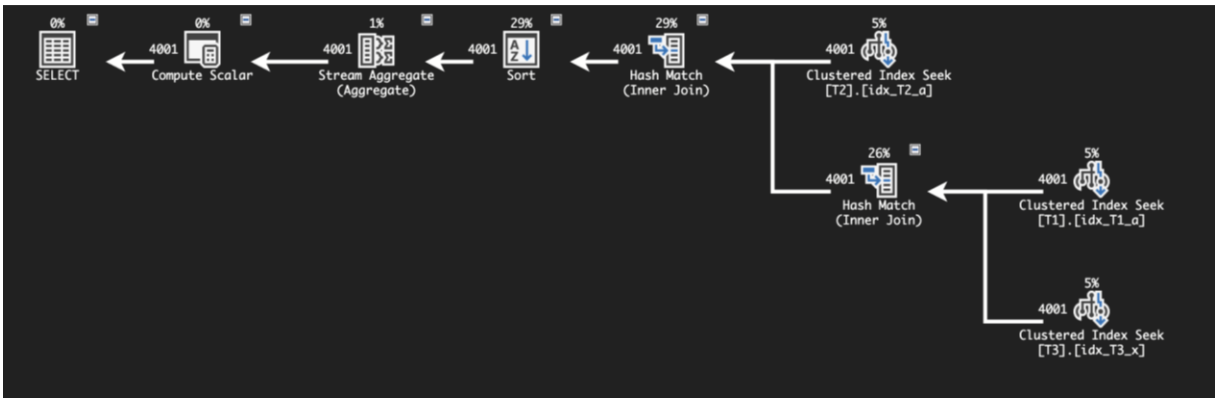
The Reason the Second Query is faster is much more subtle. So the query filters the table T1 where b > 500 and T2 where b < 3000. Here, it uses the unclustered index on T1.b in first query and then it references it to the actual table as we know with unclustered indexes. But, forcing it to scan the clustered index table may result in it not requiring the overhead time for the lookup as b > 500 is the better part of the record from the table.

Query 4:

```
SELECT
T1.a,
COUNT(T2.b) AS T2Count,
```

```
SUM(T3.z) AS TotalZ
FROM
T1
INNER JOIN
T2 ON T1.a = T2.a
INNER JOIN
T3 ON T1.a = T3.x
WHERE
T1.a BETWEEN 1000 AND 5000
GROUP BY
T1.a
GO
```

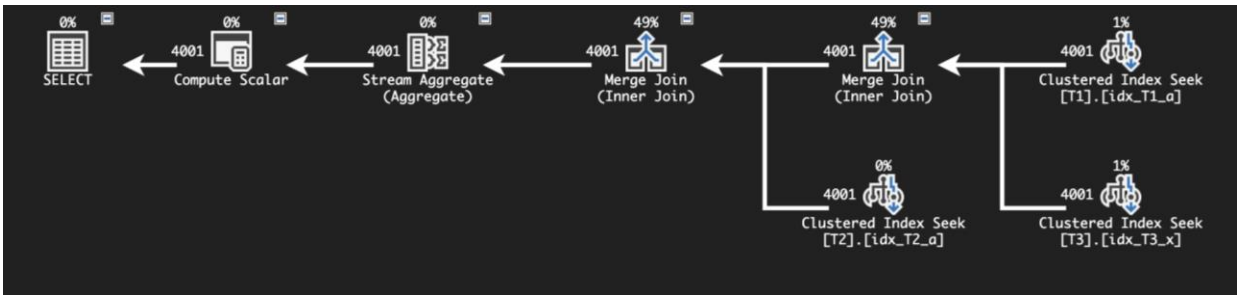
Query Plan 4:



Query 4 with hints:

```
SELECT
T1.a,
COUNT(T2.b) AS T2Count,
SUM(T3.z) AS TotalZ
FROM
T1
INNER JOIN
T2 ON T1.a = T2.a
INNER JOIN
T3 ON T1.a = T3.x
WHERE
T1.a BETWEEN 1000 AND 5000
GROUP BY
T1.a
OPTION (MAXDOP 4, MERGE JOIN)
GO
```

Optimized Query Plan 4:

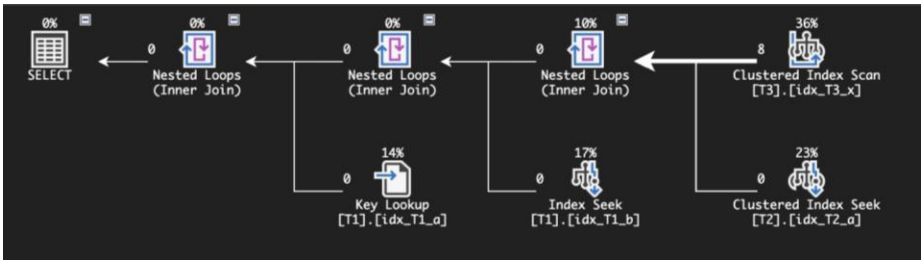


This query joins T1 with T2 and T3, which can be resource-intensive due to the number of records and the join conditions. It performs a COUNT on T2.b and a SUM on T3.z, grouped by T1.a. The filtering condition limits the result set, but the operations on the filtered data can still be significant. Setting MAXDOP 4 allows the query to use up to 4 CPU cores for parallel processing, which can speed up the execution of complex joins and aggregations. Also the Merge Join operations help in the joining by utilizing the indexes for efficient merging without the need for creating a hash table.

Query 5:

```
SELECT T1.a, T1.b, T1.c, T2.a, T2.b, T3.x, T3.y, T3.z
FROM T1
INNER JOIN T2 ON T1.b = T2.b
INNER JOIN T3 ON T2.a = T3.x
WHERE T1.c > 5000 AND T2.b < 8000 AND T3.z BETWEEN 10000 AND 20000
GO
```

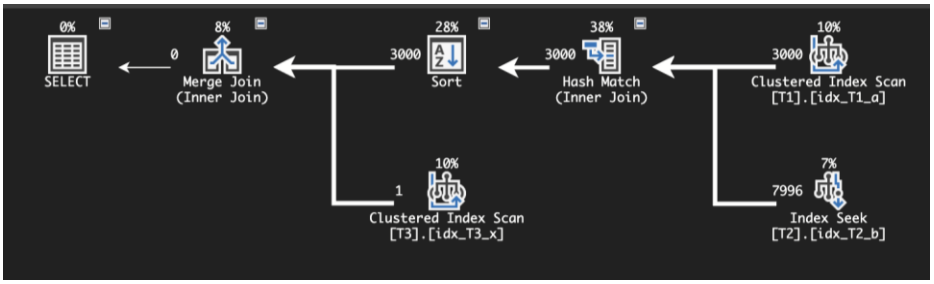
Query Plan 5:



Query 5 with hints:

```
SELECT T1.a, T1.b, T1.c, T2.a, T2.b, T3.x, T3.y, T3.z
FROM T1
INNER JOIN T2 ON T1.b = T2.b
INNER JOIN T3 ON T2.a = T3.x
WHERE T1.c > 5000 AND T2.b < 8000 AND T3.z BETWEEN 10000 AND 20000
OPTION (FORCE ORDER)
GO
```

Optimized Query Plan 5:



In this case, you might want to ensure that the joins occur in the order T1, T2, T3 to take advantage of the filters applied on each table in a specific sequence. Using the FORCE ORDER hint can ensure this join order. By forcing the join order, the optimizer will join T1 to T2 first, applying the filters on T1 and T2, and then join the result to T3, applying the final filter on T3. This can be more efficient if the join order matches the optimal execution path for the given filters and data statistics that the query optimizer maintains

Results:

	Time in ms		
	Original	**Optimized**	**% Improvement**
Query 1	**7**	**3**	**57.14**
Query 2	**19**	**15**	**21.05**
Query 3	**36**	**28**	**22.22**
Query 4	**40**	**19**	**52.5**
Query 5	**27**	**17**	**37.03**