1 Query Optimization

1.1 SeqScan

A seq scan is exploring the full table so we can simply do the following

```
SELECT * FROM unicode
```

1.2 Index Scan

An index scan is when postgres uses an index to retrieve the position of the relevant records and then fetches the record. So looking at our table we see that there is an index on codepoint so we can do the following:

```
SELECT * FROM unicode WHERE codepoint='0000' ;
```

1.3 Index Only Scan

An index only scan is similar to an index scan in the sense that postgres uses an index but postgres does not fetch the actual data from the table it simply looks at the index. So to trigger an index only scan we can retrieve the codepoint lower than some threshold.

```
SELECT * FROM unicode WHERE codepoint<'0000' ;
```

1.4 Bitmap Index Scan & Bitmap Heap Scan

A bitmap index scan means that postgres builds a bitmap using an index. The bitmap can then serves several purposes: counting, retrieve the records in order (e.g. a Bitmap Heap scan), etc.

One of the indexes on the unicode table is the index on charname. If we run the query

```
SELECT * FROM unicode WHERE charname='something' ;
```

then the query is optimized to retrieve data using the index. charname is not unique and therefore it can retrieve several tuples and when postgres estimates that many tuples will be retrieved (but not too many) it uses a bitmap. Here the only value triggering this is '<control>' as it is present 65 times.

```
SELECT * FROM unicode WHERE charname='<control>' ;
```

1.5 BitmapOr

A bitmap or is when two bitmaps are created and then we build a bitmap with the OR condition. Here it is easy to trigger this behavior using a OR condition on something already building a bitmap:

```
SELECT * FROM unicode WHERE numeric IS NOT NULL or codepoint = '0000' ;
```

1.6 BitmapAnd

In comparison the And is slightly harder to trigger. If we have conditionA AND conditionB such that one of them is very selective, the AND will favor a plan that retrieves all tuples where this condition is met. Furthermore if we use two comparison on the same column (e.g. charname => 'a' AND charname <= 'z') then the index can directly retrieve the range. Overall by choosing the right threshold we obtain the right plan:

```
SELECT * FROM unicode WHERE numeric IS NOT NULL AND charname < 'b' ;
```

1.7 Filter

A filter is very easy to trigger as it is the only way to filter a table when no indexes are present. On the unicode table, for instance, no index exists on comment so we have:

```
SELECT * FROM unicode WHERE comment IS NOT NULL ;
```

1.8 Nested Loop

A nested loop is trigger when there is a join that cannot be efficiently processed. In particular, when there are not condition on the join (thus it is more a cartesian product than a join):

```
SELECT * FROM unicode u1, unicode u2 ;
```

1.9 Hash Join and Merge Join

The hash and merge joins are two algorithms to join data. There are several reason why postgres might choose one or the other. It might depend on the type of data considered or whether the data can be retrieve in sorted order and on other criteria.

For instance, here, if we retrieve the record where numeric is not null, postgres will use an index to do so and thus retrieves then in order:

```
SELECT * FROM unicode u1, unicode u2
WHERE u1.numeric=u2.numeric
   AND u1.numeric IS NOT NULL and u2.numeric IS NULL;
```

Note that IS NOT NULL in that query is useless and NULL is never equal to NULL therefore, to trigger a merge join we can simplify this query as:

```
SELECT * FROM unicode u1, unicode u2 WHERE u1.numeric=u2.numeric ;
```

Another way of getting the merge sort is to use a datatype that postgres prefers to sort than to hash, e.g. text:

```
SELECT * FROM unicode u1, unicode u2 WHERE u1.comment=u2.comment ;
```

For the hash join it is really easy to trigger as this is generally the preferred join for postgres. For instance for an integer foreign key:

```
SELECT * FROM unicode u1, unicode u2 WHERE u1.codepoint=u2.lowercase ;
```