1.a) An efficient query plan is one that utilizes a particular resource very well. In my opinion, there are different optimizations that a query might wish to optimise for:

1. Response time – a volcano style processing system would improve response time for the first correct tuple which might be desirable, say, in a scenario where you want to kick off computation as fast as possible for example in a strategical scenario where you can’t afford to wait for all possible answers.
2. Data access
3. CPU cycles – an efficient query plan that might optimize for CPU cylces is a bulk processing style processing system where we limit function pointers which decreases the overall CPU cycles required to fulfill a query.

1.b)

Cost of executing query before migration:

We don’t know where the tuples are located, so we need to scan each part of the table in order to fetch the tuples.

1. Scan all pages of table R
   1. Scan Main:  
      # of tuples in main = 0.8 \* 5 \* 10^6 = 4 \* 10^6 tuples  
      column to scan = ‘a’  
      size of value ‘a’ = 4 Bytes  
      size of column ‘a’ = (4 \* 10^6 tuples ) \* (4 Bytes) = 1.6 \* 10^7 Bytes  
      size of column ‘a’ = (1.6 \* 10^7 Bytes) / (64 Bytes per page) = 2.5 \* 10^5 pages
   2. Scan Delta:  
      # of tuples in delta = 0.2 \* 5 \* 10^6 = 10^6 tuples  
      size of tuple = 4\*3 Bytes  
      size of delta = (10^6 tuples) \* (12 Bytes) = 1.2 \* 10^7 Bytes  
      size of delta = (1.2 \* 10^7 Bytes) / (64 Bytes per page) = 187500 pages
   3. **Cost of scan = 187500 + 250000 = 437500 pages**
2. Selection Predicate
   1. Assume that there are proportionally many a’s distributed across the table  
      # of matching a’s in main = # of tuples in main \* 0.05 = (4 \* 10^6 tuples) \* 0.05 = 200000 tuples  
      # of matching a’s in delta = # of tuples in delta \* 0.05 = (10^6 tuples) \* 0.05 = 50000 tuples
   2. Our intermediate data structure should be capable of storing 200,000 + 50,000 = 250,000 tuples  
      The result of Delta and main ‘lookup’ function is full tuples. ~~Therefore, we make the assumption that the intermediate structure is going to be full tuples, not just references. [see delta/main table lookup]~~  
      ~~Fuck that assumption, ill instead assume that the stuff from Main is by reference, and the stuff~~ from delta is by tuple. I’ll also assume that there is a header that describes when each partition of the structure starts, but for the sake of argument, this is likely to be very insignificant for the final calculations.  
      Our intermediate size = 200,000 \* (4B per reference to main) + 50,000 \* (12 Bytes per tuple) = 1,400,000 Bytes  
      Intermediate size in pages = 1,400,000 Bytes / 64 Bytes per page = 6250 Pages  
      Size of cache = 32 \* 1024 Bytes / 64 Bytes = 512 Pages  
      Since our intermediate structure is larger than the cache we need to build and load it back up again
   3. **Cost of writing and reading our structure = 2 \* 6250 = 12500 Pages**
3. Grouping attribute
   1. This will trivially fit in the cache as it is a hash table with one key and value pair. Therefore, there is 0 cost associated with probing this bad boy
   2. Once we’ve read the structure, the cost of which has been accounted for, we have to get # of matching a’s in main = 200000 tuples in order to get their value for b  
      slectivity = # number of matching a’s in main / # of tuples in main = 0.05  
      # of tuples per page in main = 4 B per value in each column / 64 Bytes per page = 16 tuples per page  
      probability of touching a page in main = 1 - (1-0.05)^16 = 0.55987333134  
      # of pages touched during fetching of b = 0.55987333134 \* # of pages in column for b = 0.55987333134 \* 2.5 \* 10^5 pages = 139968.3 pages = **139969 pages.**
4. **Total cost for running the query = 437500 + 12500 + 139969 = 589969 page faults**

Cost of executing query after migration:

1. Scan all pages of R
   1. # of tuples in R = 5 \* 10^6 tuples  
      column to scan = ‘a’  
      size of ‘a’ = 4B  
      size of column of a’s = 4 \* 5 \* 10^6 = 2 \* 10^7  
      # of pages for a = 2 \* 10^7 / 64 = **312500** pages
   2. **Cost of scan = 312500 pages**
2. Selection:
   1. The input will be just straight up the columns of a and their values, so no dereferencing cost.
   2. Number of tuples matching a = 0.05 \* 5 \* 10^6 = 250000 tuples  
      size of intermediate data structure = 250000 tuples \* 4B for address / 64 B = 15625 pages  
      This doesn’t fit in the pool, so we have a cost associated with writing and reading back:  
      2 \* 15625 = 31250 pages
3. Grouping:
   1. Cost associated with dereferencing the values:  
      selectivity = 0.05  
      # of tuples per page = 16  
      probability of touching a page = 1 - (1 – 0.05)^16 = 0.55987333134  
      # of pages touched = 0.55987333134 \* #tuples per column of b = 0.55987333134 \* 312500 = **174961 pages**
4. Total for running this query = 312500 + 31250 + 174961 = 518711 page faults

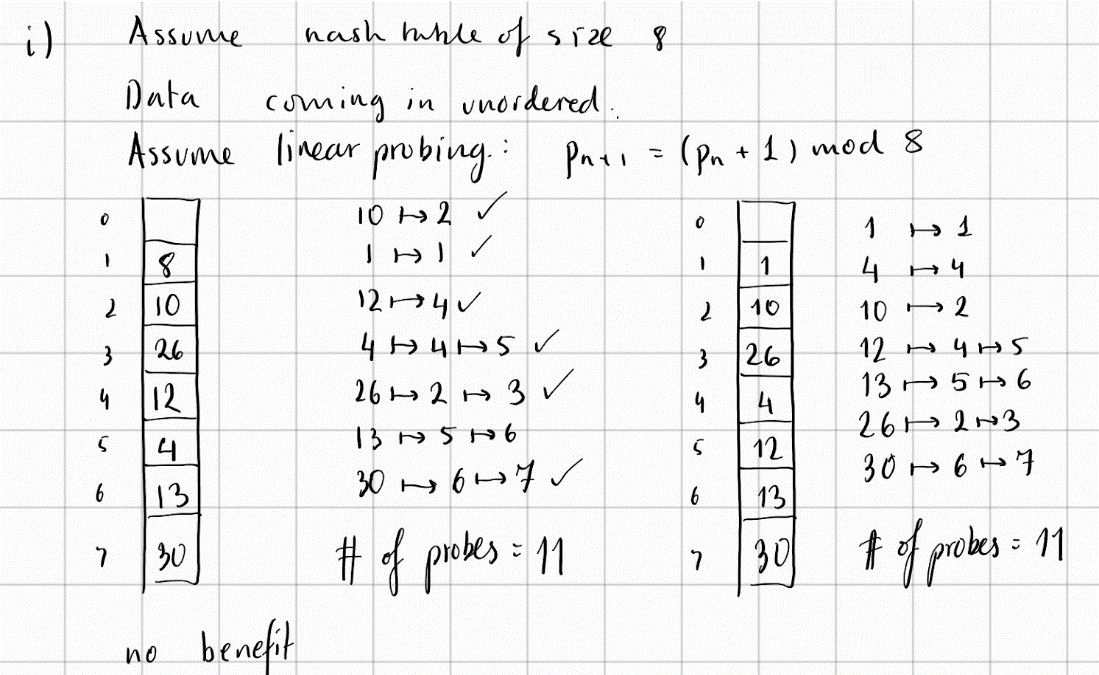
Migration cost:

1. In delta, we access data sequentially and write to main.  
   Assume that there is no cost required to shift the main storage around so that there is space to insert tuples in main.
2. Scan Delta:  
   # of tuples in Delta = 0.2 \* 5 \* 10^6 = 10^6 tuples  
   size of table = (10^6 tuples ) \* (4+4+4 Bytes) = 12000000 Bytes  
   size of table = (12000000 Bytes) / (64 Bytes per page) = **187500 pages**
3. Assume that there is no cost to split tuples up
4. We need to, for each tuple read in N-ary data insert it into a page in main, therefore, that would be 3 page faults per tuple in NAry. We can do this approximately sequentially for each tuple ~~therefore its~~ **~~187500~~** ~~page faults in order to insert into main.~~  
   Say we’re looking at one tuple, for each attribute, we page fault to bring that page into memory and write, we do this for each column. The next attribute will write to the end of the previously written one, which is still assumed to be in memory. Therefore, we should only page fault as many times as we’re required to bring in a new page, I.e. number of pages that NAry takes up.
5. Cost of migration = Read main + insert into delta = 2 \* **187,500= 3.75 \* 10^5**

Difference in page faults = 589969 – 518711 = 71258

Cost of migration / difference in page faults = 3.75 \* 10^5 / 71258 = 5.26256701 = 6

1.c.i)



1.c.ii) Ty chat gpt

When determining a good plan for a query, it is often beneficial to establish and/or maintain what is called an "interesting order". The idea is to make sure that tuples flow into operators in an order that is beneficial to the working of the operator. Name and describe 5 cases in which an interesting order can be beneficial.

Sure! Here are five cases in which an interesting order can be beneficial for query processing:

1. Grouping and Aggregation: When performing a GROUP BY or aggregation operation, it is often beneficial to sort the data by the grouping columns in advance, as this can reduce the amount of memory needed for the operation and speed up processing. For example, if we are grouping data by year and month, sorting the data by year and then by month can ensure that all the data for a given year is processed together, reducing the need to store multiple years' worth of data in memory at once.
2. Join Operations: When performing join operations, it can be beneficial to sort the data by the join key in advance, as this can help to reduce the number of disk seeks required and speed up processing. For example, if we are joining a large fact table with a smaller dimension table, sorting the fact table by the join key can ensure that all the data for a given key is processed together, reducing the need to read the dimension table multiple times.
3. Window Functions: When performing window functions, it is often beneficial to sort the data by the window partition and order columns in advance, as this can reduce the amount of memory needed for the operation and speed up processing. For example, if we are calculating a moving average over a window of 30 days, sorting the data by date can ensure that the data for each day is processed together, reducing the need to store multiple days' worth of data in memory at once.
4. Subqueries: When using subqueries, it can be beneficial to sort the data in the inner query in advance, as this can reduce the amount of memory needed for the operation and speed up processing. For example, if we are using a subquery to filter a large table, sorting the data by the filter column in the inner query can ensure that only the necessary data is processed, reducing the need to read and process the entire table.
5. Index Scans: When using index scans to retrieve data, it can be beneficial to sort the data in the order of the index, as this can help to reduce the number of disk seeks required and speed up processing. For example, if we are using an index on a column such as date, sorting the data in date order can ensure that all the data for a given date is processed together, reducing the need to read the index multiple times.

1.d) Ty chat-gpt x 2

Partitioning tables into columns and rewriting queries to provide the same results can achieve some of the benefits of decomposed storage, but not all of them. Here are some benefits that can be replicated, as well as some that cannot:

Benefits that can be replicated:

1. Improved Query Performance: By partitioning tables into columns and rewriting queries, query performance can be improved. For example, if we have a table with many columns, some of which are rarely accessed, we can partition the table into separate tables for each column, which can reduce the amount of data that needs to be read for each query.
2. Better Data Compression: Partitioning tables into columns can also improve data compression, which can reduce storage costs and improve query performance. For example, if we have a table with many columns, some of which contain repetitive data, we can partition the table into separate tables for each column, which can improve compression ratios.
3. Parallel Processing: Partitioning tables into columns can also enable parallel processing, which can further improve query performance. For example, if we have a table with many columns, we can partition the table into separate tables for each column and process each table in parallel, which can speed up query processing.

Benefits that cannot be replicated:

1. Reduced I/O: Decomposed storage reduces I/O by only reading the columns needed for a particular query. Partitioning tables into columns and rewriting queries cannot achieve the same level of I/O reduction, as all columns in a given table may still need to be read to execute a query.
2. More Efficient Aggregations: Decomposed storage enables more efficient aggregations, as columns can be pre-sorted and pre-joined before aggregation. Partitioning tables into columns and rewriting queries cannot achieve the same level of efficiency, as the necessary pre-sorting and pre-joining may still need to be performed during query execution.
3. Schema Evolution: Decomposed storage enables easier schema evolution, as adding or removing columns only requires modifying the storage layer. Partitioning tables into columns and rewriting queries cannot achieve the same level of ease, as adding or removing columns may require modifying the queries themselves.

In summary, while partitioning tables into columns and rewriting queries can achieve some of the benefits of decomposed storage, it cannot replicate all of them. Therefore, decomposed storage remains an important technique for improving query performance and reducing storage costs in database systems.