1. Due to the presence of the more duplicates in the 'cname' attribute in comparison to the those in 'headquarter' attribute, the 'headquarter' attribute is better suited to be indexed in the context of loading data into memory once the index has been created and a query is executed on the relation. Since, 'headquarter' has lesser duplicates, the index would have fewer entries and retrieval would be more efficient. The index can be leveraged to locate the relevant records quickly. With lower cardinality, the number of records to process would be reduced and would consume lesser memory. The I/O operations required to retrieve the records of interest would be reduced as well, subsequently, speeding up query processing.

2. Advantages:
   - Indexing 'timestamp' attribute would significantly improve the performance of sorting and selecting longitudinal transactions in a specific time range of interest.
   - This will also help in aggregation queries based on time intervals, thus, improving query performance
   - Indexing will also facilitate faster modifications based on the INSERT/UPDATE/DELETE with faster search with the presence of indexes.

   Disadvantages:
   - Selecting appropriate attributes for indexing is equally important. If 'timestamp' is indexed and the queries require retrieval based on other attributes, the query performance would not improve with indexing.
   - The relation undergoes thousands of modifications every second which necessitates periodic maintenance, including re-indexing or organizing indexes, with every modification, to ensure speedy query processing.
   - The attribute to be indexed will require additional storage.

3. Considering the nature of the query, we can index 'salary' attribute to speed up the query.

   We can create the index with the below:
   CREATE INDEX sal_idx ON worksFor(salary);

   The table below records the execution time of the queries for values s1 and s2 such that s1 < s2:

Records	**s1 and s2**	**Execution Time (ms)**	
		Without Index	**With Index**
100	*[s1 = 60000 and s2 = 100000]*	*0.296*	*0.061*
	[s1 = 80000 and s2 = 1100000]	*0.443*	*0.031*
1000	*[s1 = 60000 and s2 = 100000]*	*0.359*	*0.093*
	[s1 = 80000 and s2 = 1100000]	*0.509*	*0.038*
100000	*[s1 = 60000 and s2 = 100000]*	*14.278*	*3.429*
	[s1 = 80000 and s2 = 1100000]	*17.793*	*0.089*

Based on the execution times recorded at the different record level, we see that indexing 'salary' attribute does speed up query processing when the number of records in the relation increase.

4. Considering the nature of the query, we can index 'cname' attribute in the Company table and 'pid' attribute in Person table to speed up the query.

   We can create the index with the below:
   CREATE INDEX cname_idx ON Company(cname);
   CREATE INDEX pid_idx ON Person(pid);

   The table below records the execution time of the given query before and after creating the mentioned indexes:

Records	**cname**	**Execution Time (ms)**	
		Without Index	**With Index**
100	'Amazon'	*0.214*	*0.176*
	'Netflix'	*0.164*	*0.129*
	'Stripe'	*0.147*	*0.088*
1000	'Amazon'	*0.360*	*0.295*
	'Netflix'	*0.305*	*0.316*
	'Stripe'	*0.370*	*0.292*
10000	'Amazon'	*5.997*	*2.825*
	'Netflix'	*5.646*	*2.739*
	'Stripe'	*2.272*	*0.921*

   The query performance significantly improves with larger tables with the use of index.

5.

(a) B^+-trees keep balanced with almost uniform height and semi-full. With binary search performed at each level of the tree, the navigation to the interested child node takes $O(\log_n N)$ as the binary search splits the nodes that helps reach the interested node in logarithmic time.

(b) The insertion algorithm involves searching the leaf node that has at least one space available. If full, the leaf is now split into two nodes and then insert the new leaf's smallest key into parent. With this, the search can be performed in logarithmic time and splitting, and insertion of the nodes are efficient operations and take O(1), therefore, the insert time of a record is $O(\log_n N)$.

(c) Deletion algorithm involves coalescing with the neighbours and re-distribution of the keys. We begin with deleting the node and we proceed with balancing the trees with visiting the appropriate node to either occupy the parent or borrow from a sibling. With

this, because of the height of the tree and the logarithmic operation to search the node before deletion and the subsequent operations are efficient.
$O(\log_n N)$

(b) Given parameters:
Block size = 4096 bytes
Block address size = 9 bytes
Block access time = 10 ms
Record size = 200 bytes
Record key size = 12 bytes
Total records indexed = 100 million (10^8)

i. Minimum time to retrieve a record with key k will be the time it takes to traverse the tree form the root to the leaf node.

$$n \leq \frac{4096}{9 + 12}$$
$$n \leq 194$$

Considering n = 195, time can be calculated using $\lceil \log_{195} 10^8 \rceil + 1 = 5$. Since, the block access time is 10 microseconds (and 1 extra block access is required), the minimum required time is 50ms.

ii. Maximum time to retrieve a record with key k will be the time it takes to traverse all the nodes until the last node is reached where the record is available. We need to determine the height of the tree.

Height of the tree = $\lceil \log_{98}(10^8/2) \rceil = 4$
where 194/2 + 1 is 98

$$\text{Maximum time required } = (\text{height} + 1 + 1) \times 10 \text{ ms} = 60 \text{ ms}.$$

iii. With block access time = 10 microseconds, to achieve a maximum time as 80 ms, new height can be calculated as

$$80 = (height + 1 + 1) \times 10$$
$$height = 6$$

Solving, $\lceil \log_{98}(N/2) \rceil = 6$, we get $N = 2 \times 10^6$. We need 102 million records to increase maximum time by at least 20ms.

iv. With block size = 8192,

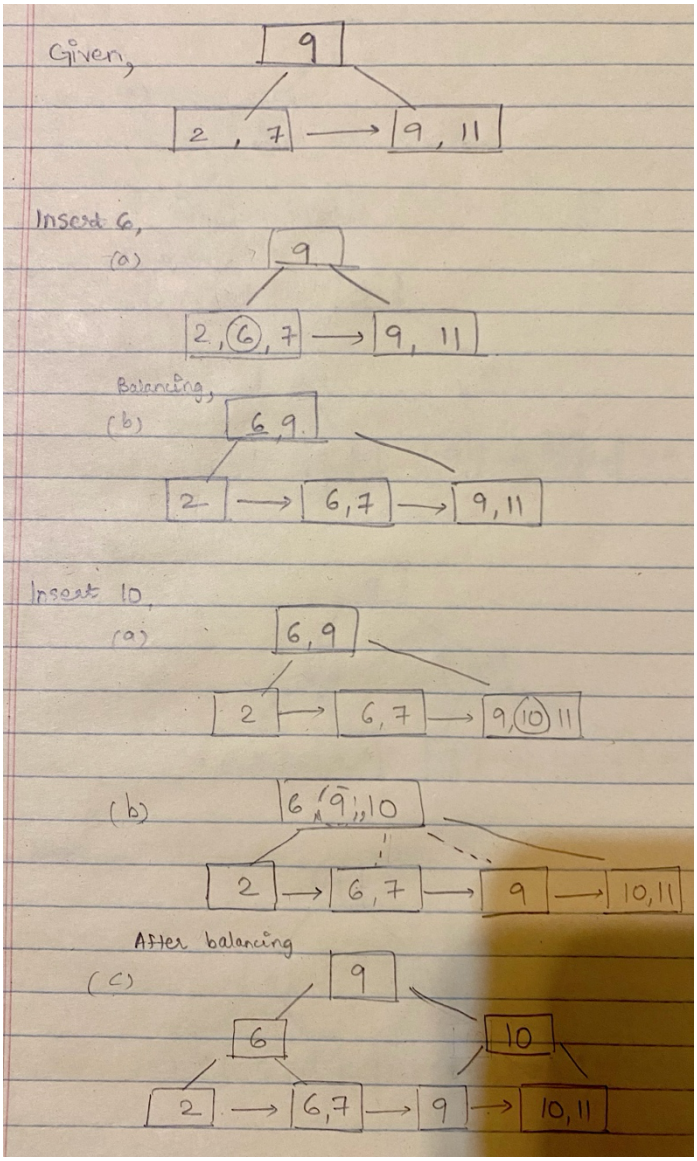
$$n \leq \frac{8192}{9 + 12}$$
$$n \leq 390$$

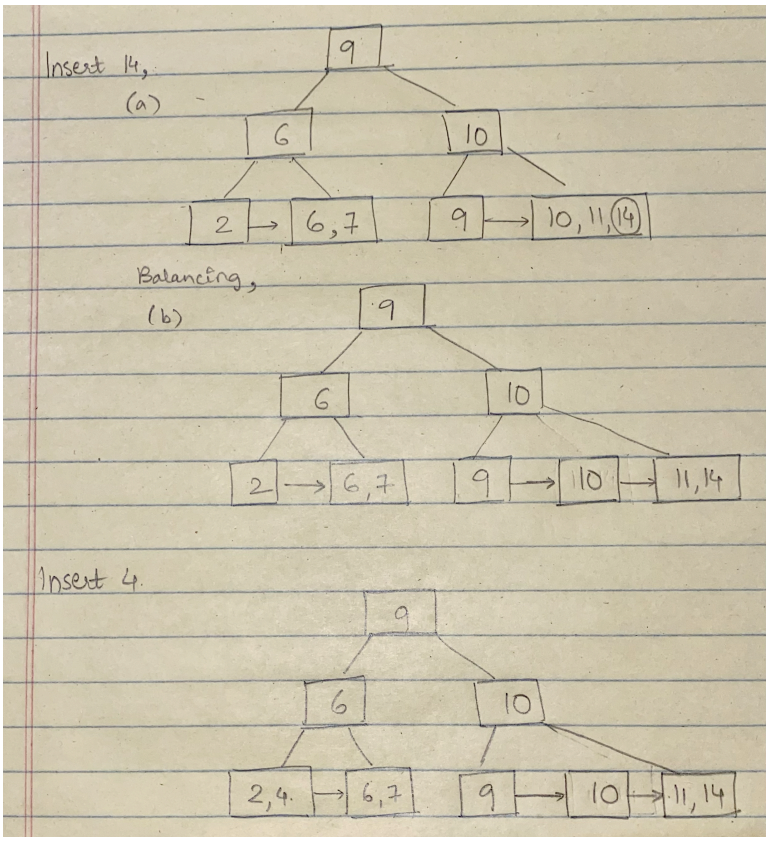
Considering n = 391, we get the new height of the tree as 5 ($\lceil \log_{195}(10.2 \times 10^8/2) \rceil + 1 = 5$).

Thus, the maximum time to retrieve a record with key k becomes $(5 + 1 + 1) \times 10 \text{ ms} = 70 \text{ ms}$. We see a decrease by 10ms if the block size is increased to 8192 bytes.

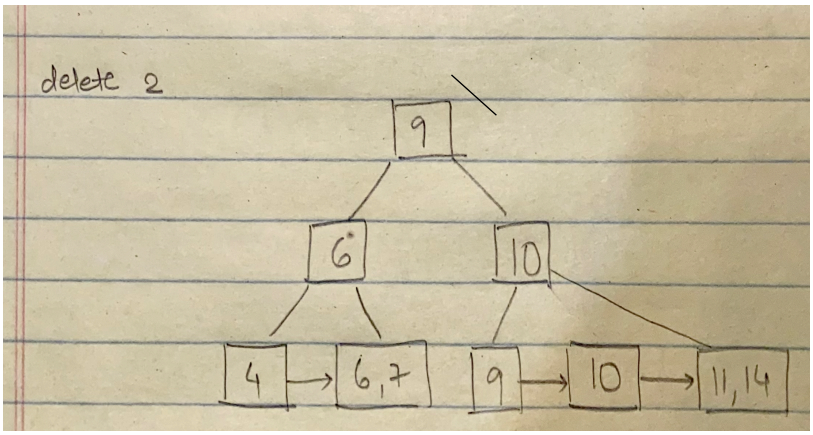
(c)

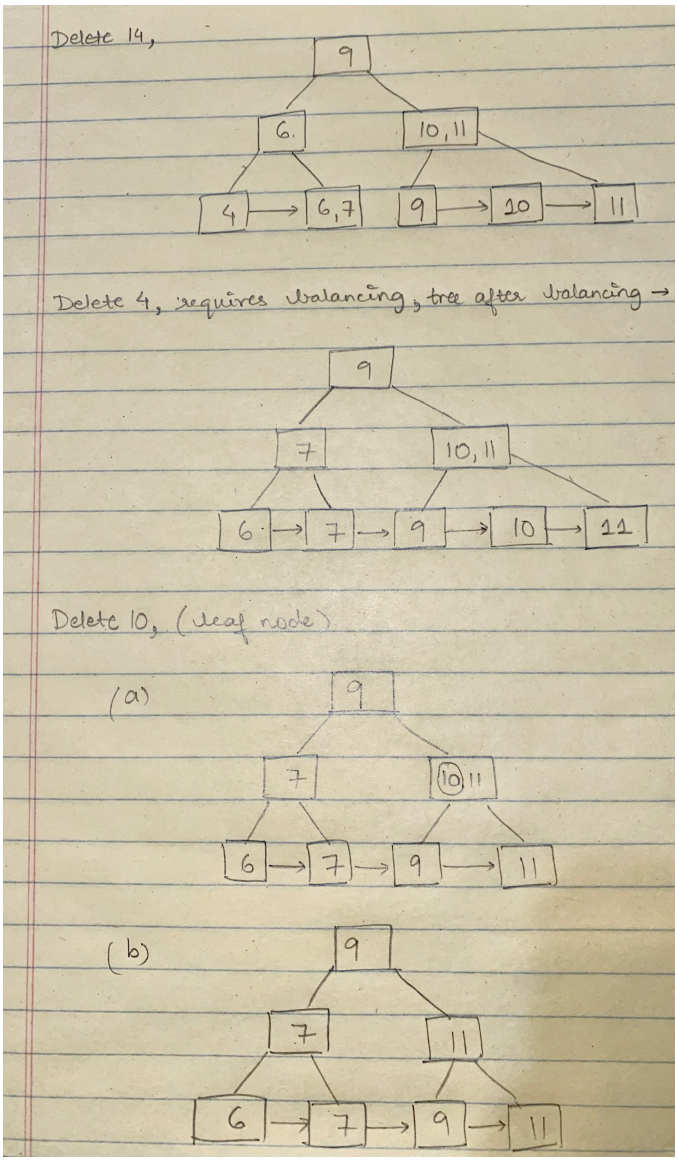
i.




Insert 14,
(a)
9
6
10
2
6,7
9
10, 11, 14
Balancing,
(b)
9
6
10
2
6,7
9
10
11,14
Insert 4.
9
6
10
2,4
6,7
9
10
11,14

ii.


delete 2
9
6
10
4
6,7
9
10
11,14



6.

(a) Binary representation of integers to insert:

0: 0000
1: 0001
2: 0010
3: 0011
4: 0100
5: 0101
6: 0110
7: 0111
8: 1000
9: 1001

Global depth: 1
Bucket local depth: 1
0110
0111

(b)

7.

(a) IO operations for nested loop:

(b)

(c)

(d)

(e)

8.

(a) Translating and optimizing the given Query Q_3,

select distinct p.a
from P p, R r1, R r2, R r3, S s
where p.a = r1.a and r1.b = r2.a and r2.b = r3.a and r3.b = S.b;

Eliminating joins, Q_4:

```
select distinct p.a
from P p
join R r1 on p.a = r1.a
join R r2 on r1.b = r2.a
join R r3 on r2.b = r3.a
join S s on r3.b = s.b;
```

(b)

P, R and S	**Q_3 (in ms)**	**Q_4 (in ms)**
10^4	32.911	26.545
10^5	306.989	208.194
10^6	60958.789	1284.796

Based on the execution time, we can say that Q_4 is the optimized in comparison to Q_3 and performs better when the size of relations increase.

9.

(a) RA equivalent after translation and optimization for the given query, Q_5:

```
select p.a
from P p
where exists (
  select 1
  from R r
  where r.a = p.a and not exists (
    select 1
    from S s
```

```
        where r.b = s.b
    )
);
```

Translating exists,

```
select p.a
from P p, R r
where p.a = r.a and not exists (
        select 1
        from S s
        where r.b = s.b
);
```

Translating not exists,

```
select distinct q.a
from (
        select p.a, r.b
        from P p, R r
        where p.a = r.a
        EXCEPT
        select r.a, r.b
        from R r, S s
        where r.b=s.b
) q;
```

Introducing joins, Q_6

```
select distinct q.a
from (
        select p.a, r.b
        from P p
        natural join R r
        EXCEPT
        select r.a, r.b
        from R r
        natural join S s
) q
order by 1;
```

(b)

P, R and S	**Q_5 (in ms)**	**Q_6 (in ms)**	**Q_7 (in ms)**
10^4	9.053	21.028	103.186
10^5	70.396	132.185	9005.639
10^6	297.874	1592.742	88501.572

Queries Q_5 and Q_6 uses hash join and hash aggregate, as compared to the subquery join for query Q_7, which is why we observe significant difference in the execution time of these queries for the given sizes of table P, R and S. Additionally, for Q_7, the array_agg() computation is slowing the query when the size of the relation increases.

Query plan for Q_5:

```
"QUERY PLAN"
"Hash Join  (cost=604.44..775.67 rows=3168 width=4) (actual time=7.611..8.927 rows=1969
loops=1)"
"  Hash Cond: (p.a = r.a)"
"  ->  Seq Scan on p  (cost=0.00..119.35 rows=6335 width=4) (actual time=0.021..0.476
rows=6335 loops=1)"
"  ->  Hash  (cost=601.94..601.94 rows=200 width=4) (actual time=7.581..7.585 rows=3109
loops=1)"
"        Buckets: 4096 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 142kB"
"        ->  HashAggregate  (cost=599.94..601.94 rows=200 width=4) (actual time=6.844..7.240
rows=3109 loops=1)"
"              Group Key: r.a"
"              Batches: 1  Memory Usage: 257kB"
"              ->  Hash Anti Join  (cost=197.82..587.44 rows=5000 width=4) (actual time=1.403..5.477
rows=3739 loops=1)"
"                    Hash Cond: (r.b = s.b)"
"                    ->  Seq Scan on r  (cost=0.00..189.00 rows=10000 width=8) (actual time=0.050..1.250
rows=10000 loops=1)"
"                    ->  Hash  (cost=119.03..119.03 rows=6303 width=4) (actual time=1.344..1.345
rows=6303 loops=1)"
"                          Buckets: 8192  Batches: 1  Memory Usage: 286kB"
"                          ->  Seq Scan on s  (cost=0.00..119.03 rows=6303 width=4) (actual
time=0.008..0.534 rows=6303 loops=1)"
"Planning Time: 0.214 ms"
"Execution Time: 9.053 ms"
```

Query Plan for Q_6:

```
"QUERY PLAN"
"HashAggregate  (cost=25422.65..25424.65 rows=200 width=4) (actual time=20.377..20.608
rows=1969 loops=1)"
"  Group Key: q.a"
"  Batches: 1  Memory Usage: 257kB"
"  ->  Subquery Scan on q  (cost=1372.76..25322.65 rows=40000 width=4) (actual
time=19.137..19.866 rows=2374 loops=1)"
```

```
"      ->  HashSetOp Except  (cost=1372.76..24922.65 rows=40000 width=12) (actual
time=19.135..19.625 rows=2374 loops=1)"
"            ->  Append  (cost=1372.76..21763.15 rows=631900 width=12) (actual
time=3.770..15.726 rows=12688 loops=1)"
"                  ->  Subquery Scan on ""*SELECT* 1""  (cost=1372.76..9323.19 rows=316750 width=12)
(actual time=3.769..8.117 rows=6427 loops=1)"
"                        ->  Merge Join  (cost=1372.76..6155.69 rows=316750 width=8) (actual
time=3.768..7.524 rows=6427 loops=1)"
"                              Merge Cond: (p.a = r.a)"
"                              ->  Sort  (cost=519.38..535.22 rows=6335 width=4) (actual time=0.846..1.562
rows=6335 loops=1)"
"                                    Sort Key: p.a"
"                                    Sort Method: quicksort  Memory: 193kB"
"                                    ->  Seq Scan on p  (cost=0.00..119.35 rows=6335 width=4) (actual
time=0.014..0.452 rows=6335 loops=1)"
"                              ->  Sort  (cost=853.39..878.39 rows=10000 width=8) (actual time=2.914..4.101
rows=10000 loops=1)"
"                                    Sort Key: r.a"
"                                    Sort Method: quicksort  Memory: 697kB"
"                                    ->  Seq Scan on r  (cost=0.00..189.00 rows=10000 width=8) (actual
time=0.029..1.074 rows=10000 loops=1)"
"                  ->  Subquery Scan on ""*SELECT* 2""  (cost=1370.19..9280.46 rows=315150 width=12)
(actual time=3.157..6.860 rows=6261 loops=1)"
"                        ->  Merge Join  (cost=1370.19..6128.96 rows=315150 width=8) (actual
time=3.156..6.293 rows=6261 loops=1)"
"                              Merge Cond: (s.b = r_1.b)"
"                              ->  Sort  (cost=516.81..532.56 rows=6303 width=4) (actual time=0.817..1.171
rows=6303 loops=1)"
"                                    Sort Key: s.b"
"                                    Sort Method: quicksort  Memory: 193kB"
"                                    ->  Seq Scan on s  (cost=0.00..119.03 rows=6303 width=4) (actual
time=0.016..0.443 rows=6303 loops=1)"
"                              ->  Sort  (cost=853.39..878.39 rows=10000 width=8) (actual time=2.333..3.255
rows=10000 loops=1)"
"                                    Sort Key: r_1.b"
"                                    Sort Method: quicksort  Memory: 697kB"
"                                    ->  Seq Scan on r r_1  (cost=0.00..189.00 rows=10000 width=8) (actual
time=0.038..0.752 rows=10000 loops=1)"
"Planning Time: 0.373 ms"
```

```
"Execution Time: 21.028 ms"
```

Query plan for Q_7:

```
"QUERY PLAN"
"Subquery Scan on nestedr  (cost=1491.80..7863.47 rows=199 width=4) (actual
time=9.573..103.088 rows=1969 loops=1)"
"  InitPlan 2 (returns $1)"
"    ->  Result  (cost=119.03..119.04 rows=1 width=32) (actual time=1.024..1.024 rows=1
loops=1)"
"          InitPlan 1 (returns $0)"
"            ->  Seq Scan on s  (cost=0.00..119.03 rows=6303 width=4) (actual time=0.016..0.606
rows=6303 loops=1)"
"  ->  GroupAggregate  (cost=1372.76..7742.44 rows=199 width=36) (actual time=9.572..102.990
rows=1969 loops=1)"
"        Group Key: p.a"
"        Filter: (NOT (array_agg(r.b) <@ $1))"
"        Rows Removed by Filter: 2072"
"        ->  Merge Join  (cost=1372.76..6155.69 rows=316750 width=8) (actual time=8.457..10.135
rows=6427 loops=1)"
"              Merge Cond: (p.a = r.a)"
"              ->  Sort  (cost=519.38..535.22 rows=6335 width=4) (actual time=1.443..1.649 rows=6335
loops=1)"
"                    Sort Key: p.a"
"                    Sort Method: quicksort  Memory: 193kB"
"                    ->  Seq Scan on p  (cost=0.00..119.35 rows=6335 width=4) (actual time=0.016..0.809
rows=6335 loops=1)"
"              ->  Sort  (cost=853.39..878.39 rows=10000 width=8) (actual time=7.007..7.536
rows=10000 loops=1)"
"                    Sort Key: r.a"
"                    Sort Method: quicksort  Memory: 697kB"
"                    ->  Seq Scan on r  (cost=0.00..189.00 rows=10000 width=8) (actual time=0.048..2.811
rows=10000 loops=1)"
"Planning Time: 0.278 ms"
"Execution Time: 103.186 ms"
```

10. Below is the timing table for scanning and sorting a bag of size *n*

size *n* of relation S	avg execution time to scan S (in ms)	avg execution time to sort S (in ms)

10^1	0.014	0.029
10^2	0.020	0.040
10^3	0.076	0.168
10^4	0.682	1.371
10^5	6.417	13.963
10^6	47.772	92.181
10^7	351.328	1067.711
10^8	5062.151	8975.180

(a) Based on the execution times for scanning and sorting S, it is evident that sorting requires more operations and resources based on the size of S. Scanning is also sequential for the sizes listed above and for sorting, the sort method changed based on the size of S.

n = 10^3	n = 10^8
Scanning: "QUERY PLAN" "Seq Scan on s (cost=0.00..9.39 rows=639 width=4) (actual time=0.007..0.049 rows=639 loops=1)" "Planning Time: 0.153 ms" "Execution Time: 0.076 ms"	*Scanning:* "QUERY PLAN" "Seq Scan on s (cost=0.00..992881.75 rows=71319675 width=4) (actual time=0.028..3857.758 rows=63208603 loops=1)" "Planning Time: 3.112 ms" "Execution Time: 5062.151 ms"
Sorting: "QUERY PLAN" "Sort (cost=39.17..40.76 rows=639 width=4) (actual time=0.098..0.135 rows=639 loops=1)" " Sort Key: x" " Sort Method: quicksort Memory: 25kB" " -> Seq Scan on s (cost=0.00..9.39 rows=639 width=4) (actual time=0.010..0.057 rows=639 loops=1)" "Planning Time: 0.049 ms" "Execution Time: 0.168 ms"	*Sorting:* "QUERY PLAN" "Gather Merge (cost=4510309.15..10656019.09 rows=52673836 width=4) (actual time=3205.466..7702.490 rows=63208603 loops=1)" " Workers Planned: 2" " Workers Launched: 2" " -> Sort (cost=4509309.12..4575151.42 rows=26336918 width=4) (actual time=3167.624..3861.916 rows=21069534 loops=3)" " Sort Key: x" " Sort Method: external merge Disk: 248416kB" " Worker 0: Sort Method: external merge Disk: 248368kB" " Worker 1: Sort Method: external merge Disk: 245744kB" " -> Parallel Seq Scan on s (cost=0.00..543054.18 rows=26336918 width=4) (actual time=0.033..880.699 rows=21069534 loops=3)" "Planning Time: 0.234 ms"

	"Execution Time: 8975.180 ms"

(b) Based on the query plan obtained using “explain analyze”, we observe that quick sort is used when n is smaller as it fits in the available memory, therefore, the execution time does not significantly increase with n. When n becomes larger, external merge sort is used to sort the relation as larger memory is required, thereby, increasing the I/O operations as well as the execution time. Based on the size, operations can be executed parallelly by multiple workers, and the output is combined.

11.

(a) *Query (Scan):*

```
EXPLAIN ANALYZE
   SELECT DISTINCT x FROM S;
```

Query (Sort):

```
EXPLAIN ANALYZE
   SELECT DISTINCT x FROM S
   ORDER BY 1;
```

	Query (Scan): *EXPLAIN ANALYZE* *SELECT DISTINCT x FROM S;*	*Query (Sort):* *EXPLAIN ANALYZE* *SELECT DISTINCT x FROM S* *ORDER BY 1;*
size n of relation S	avg execution time to scan S (in ms)	avg execution time to sort S (in ms)
10^1	0.038	0.068
10^2	0.070	0.093
10^3	0.310	0.548
10^4	3.747	6.749
10^5	33.339	39.190
10^6	323.219	319.134
10^7	3237.434	3266.858
10^8	37336.631	34115.305

(a) *Query (Scan):*

```
EXPLAIN ANALYZE
   SELECT DISTINCT x FROM S
   GROUP BY 1;
```

Query (Sort):

```
EXPLAIN ANALYZE
   SELECT DISTINCT x FROM S
   GROUP BY 1
   ORDER BY 1;
```

size n of relation S	avg execution time to scan S (in ms)	avg execution time to sort S (in ms)
10^1	0.048	0.058
10^2	0.087	0.089
10^3	0.570	0.702
10^4	4.758	9.041
10^5	41.585	44.332

10^6	320.181	322.192
10^7	2716.843	2665.578
10^8	28849.012	28917.935

(c) As n increases, the query plan changes from a simple sequential scan to quicksort and merge sort for the distinct and the group by queries.

Query plans for "distinct" query:

n = 10^3

```
"QUERY PLAN"
"HashAggregate  (cost=41.88..43.88 rows=200 width=4) (actual time=0.205..0.272 rows=618 loops=1)"
"  Group Key: x"
"  Batches: 1  Memory Usage: 89kB"
"  ->  Seq Scan on s  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.013..0.062 rows=618
loops=1)"
"Planning Time: 0.144 ms"
"Execution Time: 0.310 ms"
```

n = 10^6

```
"QUERY PLAN"
"Unique  (cost=8484.75..8485.75 rows=200 width=4) (actual time=251.648..309.486 rows=632959
loops=1)"
"  ->  Sort  (cost=8484.75..8485.25 rows=200 width=4) (actual time=251.647..269.998 rows=632959
loops=1)"
"        Sort Key: x"
"        Sort Method: external merge  Disk: 7448kB"
"        ->  Gather  (cost=8455.11..8477.11 rows=200 width=4) (actual time=77.528..160.273
rows=632959 loops=1)"
"              Workers Planned: 1"
"              Workers Launched: 1"
"              ->  HashAggregate  (cost=7455.11..7457.11 rows=200 width=4) (actual time=74.416..135.907
rows=316480 loops=2)"
"                    Group Key: x"
"                    Batches: 21  Memory Usage: 10305kB  Disk Usage: 7280kB"
"                    Worker 0:  Batches: 21  Memory Usage: 10305kB  Disk Usage: 7144kB"
"                    ->  Parallel Seq Scan on s  (cost=0.00..6524.29 rows=372329 width=4) (actual
time=0.011..13.810 rows=316480 loops=2)"
"Planning Time: 0.065 ms"
"Execution Time: 323.219 ms"
```

For sorting queries, the execution time changes due to the additional in-memory scans and external memory I/O operations and passes.

Query plan for Sorting after distinct for n = 10^6:

```
"QUERY PLAN"
"Unique  (cost=8484.75..8485.75 rows=200 width=4) (actual time=251.648..309.486 rows=632959 loops=1)"
"  ->  Sort  (cost=8484.75..8485.25 rows=200 width=4) (actual time=251.647..269.998 rows=632959 loops=1)"
"        Sort Key: x"
"        Sort Method: external merge  Disk: 7448kB"
"        ->  Gather  (cost=8455.11..8477.11 rows=200 width=4) (actual time=77.528..160.273 rows=632959 loops=1)"
"              Workers Planned: 1"
"              Workers Launched: 1"
"              ->  HashAggregate  (cost=7455.11..7457.11 rows=200 width=4) (actual time=74.416..135.907 rows=316480 loops=2)"
"                    Group Key: x"
"                    Batches: 21  Memory Usage: 10305kB  Disk Usage: 7280kB"
"                    Worker 0:  Batches: 21  Memory Usage: 10305kB  Disk Usage: 7144kB"
"                    ->  Parallel Seq Scan on s  (cost=0.00..6524.29 rows=372329 width=4) (actual time=0.011..13.810 rows=316480 loops=2)"
"Planning Time: 0.065 ms"
"Execution Time: 323.219 ms"
```

Query plan for Sorting after group-by for n = 10^6:

```
"QUERY PLAN"
"Unique  (cost=8464.76..8488.76 rows=200 width=4) (actual time=202.277..308.233 rows=632959 loops=1)"
"  ->  Group  (cost=8464.76..8488.26 rows=200 width=4) (actual time=202.276..273.118 rows=632959 loops=1)"
"        Group Key: x"
"        ->  Gather Merge  (cost=8464.76..8487.76 rows=200 width=4) (actual time=202.275..245.310 rows=632959 loops=1)"
"              Workers Planned: 1"
"              Workers Launched: 1"
"              ->  Sort  (cost=7464.75..7465.25 rows=200 width=4) (actual time=166.443..176.219 rows=316480 loops=2)"
"                    Sort Key: x"
"                    Sort Method: external merge  Disk: 4360kB"
```

```
"               Worker 0:  Sort Method: external merge  Disk: 3088kB"
"               ->  Partial HashAggregate  (cost=7455.11..7457.11 rows=200 width=4) (actual
time=72.426..122.573 rows=316480 loops=2)"
"                     Group Key: x"
"                     Batches: 21  Memory Usage: 10305kB  Disk Usage: 7528kB"
"                     Worker 0:  Batches: 5  Memory Usage: 10561kB  Disk Usage: 3800kB"
"                     ->  Parallel Seq Scan on s  (cost=0.00..6524.29 rows=372329 width=4) (actual
time=0.006..12.561 rows=316480 loops=2)"
"Planning Time: 0.072 ms"
"Execution Time: 322.192 ms"
```