Assignment 3
120 points

Professor Yanlei Diao

Question 1 [12 points] Sorting

Suppose that you have a file with 1,000,000 pages and you have 21 buffer pages. Answer the following questions assuming that the external sorting algorithm is used.

(1) How many runs will you produce in the first pass?

(2) How many passes will it take to sort the file completely?

(3) What is the total I/O cost of sorting the file?

(4) How many buffer pages do you need to sort the file completely in just two passes?

Answer:

(1) In the first pass (Pass 0), N/B runs of B pages each are produced, where N is the number of file pages and B is the number of available buffer pages:
$\lceil 1000000/21 \rceil = 47620$.

(2) The number of passes required to sort the file completely, including the initial sorting pass, is $\lceil \log_{B-1} N/B \rceil + 1 = \lceil \log_{20} 47620 \rceil + 1 = 5$

(3) Since each page is read and written once per pass, the total number of page I/Os for sorting the file is 2 * N * (#passes) = 2 * 1,000,000 * 5 = 10,000,000

(4) In Pass 0, N/B runs are produced. In Pass 1, we must be able to merge this many runs; i.e. $B - 1 \geq N/B$. This implies that B must at least be large enough to satisfy $B(B - 1) \geq N$; this can be used to guess B, and the guess must be validated by checking the first inequality. In this case we need at least 1001 pages.

Question 2 [20 points] Evaluation of selections

Consider a relation with this schema:

Employees(<u>*eid*: integer</u>, *ename*: string, *sal*: integer, *title*: string, *age*: integer)

Suppose that the following indexes, all using Alternative (2) for data entries, exist:

a) a hash index on *eid*,

b) a B+ tree index on *sal*,

c) a hash index on *age*,

d) a clustered B+ tree index on <*age*, *sal* >.

Each Employees record is 100 bytes long, and you can assume that each index data entry is 20 bytes long. The Employees relation contains 10,000 pages and each page holds 20 data records.

Consider each of the following selection conditions. Assume that the *reduction factor* (RF) for each term that matches an index is 0.1, except those terms on the primary key. Compute the cost of the *most selective access path*, including the file scan and various index scans, for retrieving all Employees tuples that satisfy the condition:

(1) *sal* > 100

(2) *age* = 25

(3) *eid* = 1000

(4) *sal* > 200 ∧ age > 20

Answer:
For this problem, we are given a relation (file) of 10,000 pages. A B+ tree built on this relation will have around 10,000*(20bytes / 100bytes) = 2000 leaf pages. Considering an average fanout of 133, the B+ tree will have three levels including the leaf pages.

(1) *sal* > 100.
For this condition, a file scan would probably be best, since a clustered index does not exist on sal. Using the unclustered index would accrue the following cost:

2 (B+tree top-down traversal) +
10,000 pages * (20bytes / 100bytes) * 0.1 (B+tree leaf page scan) +
min(10,000 pages * 20 tuples per page * 0.1, 10000) (retrieval of data records)
= 10202,

It would be inferior to the file scan cost of 10000.

(2) *age* = 25.
The clustered B+ tree index would be the best option here, with a cost of

2 (lookup) +
10,000 * 0.2 (reduction) * 0.1 +
10000 pages * 0.1 (selectivity)
= 1202.

Although the hash index has a lesser lookup time, the potential number of record lookups

min(10000 pages * 0.1 * 20 tuples per page = 20000, 10000)

renders the clustered index more efficient.

(3) *eid* = 1000.
Since *eid* is a candidate key, one can assume that only one record will be in each bucket of the hash table on *eid*. Thus, the total cost is roughly

1.2 (lookup) + 1 (record access), which is 2 or 3.

(4) *sal* > 200 ∧ age > 20.
We can use the clustered B+tree on <*age*, *sal*> if the *age* > 20 clause is examined first
(in this case, the two terms, *sal* > 200 ∧ age > 20, form a prefix of the B+tree search key).

The cost of scanning B+ tree leaf pages is affected only by the reduction factor of *age* > 20, as after the B+tree top down traversal leads us to the first page with *age* = 21, we need to scan all subsequent leaf pages to locate data entries with a greater age and a matching salary.

The cost of retrieving data records from the relation (almost sorted by <*age*, *sal*>) varies, depending on how the data records matching *sal* > 200 ∧ age > 20 are laid out across pages. While there may be a special case that the reduction factor for I/O can reach 0.1*0.1, we would expect more common cases that the reduction factor is just 0.1 (only for *age* > 20).

So the total cost using this B+tree is the same as that of *age* = 25 (in this contrived example!).

Note: For this particular problem, we also accept the answer using 0.1*0.1 in the calculation (although it is less unlikely) because some optimizer may (rather naively) use the product of the reduction factors of these two terms to estimate the cost.

Question 3 [10 points] Output of join algorithms
Suppose we have two unary (one attribute only) relations, R and S:

R	S
7	8
2	4
9	2
8	1
3	3
9	2
1	7
3	3
6	

Show the result of joining R and S using each of the following algorithms. List the results in the order that they would be output by the join algorithm. Note that the result relation contains *only one attribute*, which is the common attribute between R and S.

(1) Sort merge join.

(2) Hash join. Assume there are two hash buckets, numbered 0 and 1, and that the hash function sends even values to bucket 0 and odd values to bucket 1. In Phase II, use R as the "build" relation and S as the "probe" relation. Assume that bucket 0 is read first and that the contents of a bucket are read in the same order as they were written.

Answer:
(1) The results come out in sorted order on the join attribute.
1, 2, 2, 3, 3, 3, 3, 7, 8

(2) After first phase:
R bucket 0: 2, 8, 6
R bucket 1: 7, 9, 3, 9, 1, 3
S bucket 0: 8, 4, 2, 2
S bucket 1: 1, 3, 7, 3
First load R bucket 0 and stream S bucket 0 to find matches: 8, 2, 2
Then load R bucket 1 and stream S bucket 1 to find matches: 1, 3, 3, 7, 3, 3
Final result:
8, 2, 2, 1, 3, 3, 7, 3, 3

Question 4 [32 points] Improved Hash Algorithms

Hybrid Hash Join. Consider a join of relations R and S using hash join, where $M = num\_pages\_in\_R = 400$ pages and $N = num\_pages\_in\_S = 500$ pages. Assume that we have B = 40 pages of memory available as our buffer. We observe that there is more than enough memory to run the two-phase hash join algorithm, so we try to keep one of the hash buckets in memory to avoid writing it during the partitioning phase and then re-reading it during the probing phase. Assuming that all buckets have the same size.

(1) [6 points] Do we have enough memory to perform the hybrid hash join still in two phases (or two passes)?

(2) [6 points] How many buckets should be used to perform the join in two phases?

(3) [4 points] What would be the cost in number of I/O operations in the improved join algorithm?

Advanced hash algorithms for aggregation. Consider the following query:

```
Select count(*)
From  R
Group By R.a
```

Again, R has 400 pages, and the memory size is 40 pages.

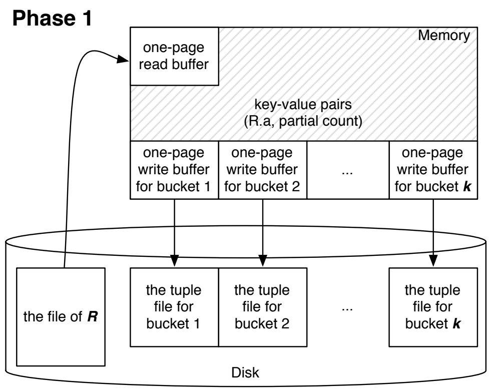
(4) [6 points] Now apply the idea of hybrid hash to group by-aggregation by still keeping the first hash bucket in memory. Compute the I/O cost of answering the above query using the new algorithm.

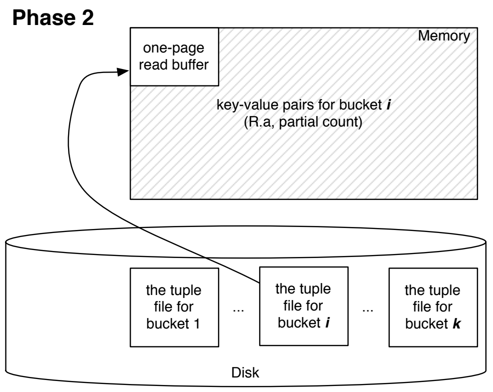
(5) [10 points] There exists a more advanced hash algorithm for group by-aggregation. Given the above query, it works as follows:

- As we read tuples in R, we build an in-memory data structure H that maps each distinct value of R.a (the key of H) to the partial count (the value of H).
- As more tuples are read, the partial counts of existing keys in H can be updated and new key-value pairs can be added to H.
- Let us set the maximum size of H to be B_H, which is less than the total memory size B. Once the map H reaches its maximum size, there are two possible cases of dealing with each tuple: (a) If R.a already exists in H, we simply update the partial count. (b) Otherwise, we hash the tuple to one of k buckets, place the tuple in the write buffer of that bucket (just 1 page), and flush the write buffer when it becomes full.
- After we have read all tuples from R, we already have the counts for a subset of values of R.a. To complete the computation for other values of R.a, we read each bucket back, one at a time, and perform any in-memory processing needed for the group by-aggregation in this bucket.

Assume that all distinct values of R.a and their counts can be held in U pages.

- What is the I/O cost of the query if the memory size B is larger than U?
- What is the minimum memory size that allows the query to be completed by reading R at most twice?


Phase 1
Memory
one-page read buffer
key-value pairs (R.a, partial count)
one-page write buffer for bucket 1
one-page write buffer for bucket 2
...
one-page write buffer for bucket k
the file of R
the tuple file for bucket 1
the tuple file for bucket 2
...
the tuple file for bucket k
Disk


Phase 2
Memory
one-page read buffer
key-value pairs for bucket i (R.a, partial count)
the tuple file for bucket 1
...
the tuple file for bucket i
...
the tuple file for bucket k
Disk

Answer.

(1) Consider the smaller relation R. In the partitioning phase, hash R tuples into k buckets, and keep one bucket of R (say R_0) in memory rather than staging it to disk.

Size of bucket R_0 = M/k, which is kept in memory.
Number of buckets written to disk = k – 1.
Memory needed for 1st pass = 1 input buffer + 1 bucket R0 + (k – 1) output buffers for other buckets:

$$1 + M/k + (k - 1) = k + M/k \le B$$

We can rewrite the above inequality as:

$$k^2 - B*k + M \le 0, \text{ or}$$
$$(k-B/2)^2 + (M-B^2/4) \le 0$$

If the above inequality holds, then we have must have $(M-B^2/4)$, or $B \ge 2\sqrt{M}$. In our example, M =400, B=40, so we just have enough memory to satisfy the above constraint.

(2) In the complete algorithm, we first partition S as before. When we partition R, we keep R_0 in memory. So when the second probing phase starts, we already have R_0 in memory.

The normal two-phase hash join cost = 3(M + N).

By simply keeping R_0 in memory, we avoid writing & re-reading this bucket, so we save $2* |R_0|$ I/Os.

$$\begin{aligned}\text{Cost} &= \text{normal hash join cost} - 2*|R_0| \\ &= 3(M + N) - 2*(M/k)\end{aligned}$$

To minimize the above cost, we need to minimize k such that $(k-B/2)^2 + (M-B^2/4) \leq 0$ still holds. In this example, $k = B/2 = 20$.

(3) So the minimum cost is:

$$\begin{aligned}\text{Cost} &= \text{normal hash join cost} - 2*|R_0| \\ &= 3(M + N) - 2*(M/k) \\ &= 3(400 + 500) - 2*(400/20) \\ &= 2700 - 40 \\ &= 2660\end{aligned}$$

(4) All I/O related operations run as above for the relation R. So the I/O cost simply is:

$3*M - 2*|R_0| = 1500-40=1100$.

(5) In the first phase, $B - k - 1$ pages of value-count pairs are kept in memory. Apparently, given a memory size B, k needs to satisfy the following inequality: $k \leq B - 1$.

$U - (B - k - 1)$ pages of value-count pairs are partitioned into k on-disk buckets. Assuming they are evenly partitioned, each on-disk bucket requires $(U - (B - k - 1))/k$ pages of memory to hold value-count pairs in the second phase. We can have another inequality according to the memory allocation in the second phase: $(U - (B - k - 1))/k \leqslant B - 1$. Then, given U and B, k needs to satisfy $k \geqslant (U - (B - 1)) / (B-2)$.

Now we have two constraints for k. When there exists a value for k to satisfy both constraints, B and U have to satisfy $(U - (B - 1)) / (B-2) \leqslant B - 1$. So $U - (B - 1) \leqslant (B - 1)(B - 2)$. Thus, $U \leqslant (B-1)(B-1)$. We can get the minimum size of B: $B \geqslant \sqrt{U} + 1$.

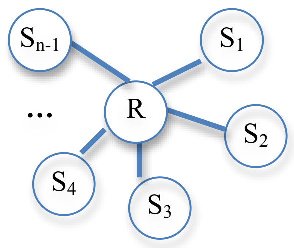
Question 5 [16 points] Complexity of Query Optimization

Analyze the complexity of System R-style query optimization:

(1) [10 points] What is the complexity of dynamic programming for finding an optimal plan for an *n*-way join? Here, consider the total number of plans that the query optimizer enumerates and denote the complexity using the big-O notation.

(2) [6 points] What is the maximum number of plans stored in an intermediate pass?

Hint: consider the star join graph as shown below and analyze the complexity related to this graph. Consider System R-style query optimization, as used in PostgreSQL, for a query involving *n* tables and the following join relationships, where an edge denotes a join predicate between two tables:



Recall that System R-style query optimizer has the following features:

- It considers only left-deep plans.
- It avoids cross products whenever possible (i.e., considers first joins with predicates and then those without join predicates).
- It uses dynamic programming to find optimal subplans involving 1 table, 2 tables, …, up to *n* tables (which are referred to as pass 1, pass 2, …, pass *n* of the dynamic programming procedure).

Answer:

(1) To get the total number of plans considered, we need to analyze each pass. Without loss of generalization, let's consider the pass *k*. Starting from Pass 2, the number of plans considered is equal to the number of *k*-relation subsets times the number of choices of the right relation from the selected subset.

The number of *k*-relation subsets is $\binom{n-1}{k-1}$ and the number of choices of the right relation is *n*-*k*.

So the pass *k* considers (*n*-*k*) $\binom{n-1}{k-1}$ plans. The total number of plans considered is at most:

$$\sum_{k=1}^{n-1}(n-k)\binom{n-1}{k-1} \le n\sum_{k=1}^{n-1}\binom{n-1}{k-1} = O(n2^{n-1})$$

(2) For pass *k*, we only need to store at most $\binom{n-1}{k}$ plans.

Maximizing it, we can get:

$$k = \lceil \tfrac{n-1}{2} \rceil .$$

So the maximum number of plans stored in one pass is

$$\binom{n-1}{\lceil \frac{n-1}{2} \rceil} .$$

Question 6 [30 points]: Query Optimization in PostgreSQL

In this problem set, we consider a publication database containing information of over 3 million papers published in computer science conferences and journals (this data was derived from the DBLP system, maintained by Michael Ley at http://www.informatik.uni-trier.de/_ley/db/).

* **Schema**. This database consists of five tables: (1) an authors table, containing the names of authors, (2) the venue table, containing information about conferences or journals where papers are published, (3) the papers table, describing the papers themselves, (4) the paperauths table which indicates which authors wrote which papers, and (5) the papertypes . The schema is the following:

```
authors (id: INTEGER NOT NULL, name: VARCHAR(200))

venue (id: INTEGER NOT NULL, name: VARCHAR(200) NOT NULL, year: INTEGER NOT NULL, school:
VARCHAR (200), volume: VARCHAR(50), number: VARCHAR(50), type: INTEGER NOT NULL)

papers (id: INTEGER, name: VARCHAR NOT NULL, venue: INTEGER REFERENCES VENUE(id),
pages: VARCHAR(50), url: VARCHAR);

paperauths (paperid: INTEGER REFERENCES PAPERS(id),authid: INTEGER REFERENCES AUTHORS(id))

papertypes (id: INTEGER NOT NULL, type: VARCHAR(20) NOT NULL)
```

* **Indexes**. Your database is allowed and only allowed to have the following indexes:

- All primary key indexes,
- A B+ tree on the <name> attribute of the **authors** table, and
- A B+ tree on the <paperid, authid> attributes of the **paperauths** table.

Recall that the command to create an index, e.g., for the last index, is:

```
create index authors_name on authors (name);
create index paperauths_pkey on paperauths (authid,paperid);
```

If you have created other indexes, please drop them using the `drop index 'index-name'` command.

* **System catalog and statistics.** As we learned in class, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans.

One important type of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table **pg_class**, in the columns reltuples and relpages. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'author%';
```

When a table is large, the number of tuples in the pg_class may differ slightly from the actual number of tuples (which you can compute using "Select count(*) From table_name"). For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time.

The view **pg_stats** provides additional information about each attribute. Of particular importance are:

- *avg_width*: the average width of an attribute, which is especially useful for variable length attributes.

- *n_distinct*: the number of distinct values in an attribute. If greater than zero, it is the estimated number of distinct values in the column. If it is -1, then this attribute is declared to be unique so it has a distinct value in each row.
- *histogram_bounds*: and the histogram bounds that divide the column's values into groups of approximately equal population.

```
select tablename, attname, avg_width, n_distinct, histogram_bounds
from pg_stats
where tablename like 'author%';
```

Other useful commands:

mysql: **`SHOW TABLES`**
postgresql: **`\d`**

mysql: **`SHOW INDEXES`**
postgresql: **`\di`**

mysql: **`SHOW COLUMNS`**
postgresql: **`\d` table**

mysql: **`DESCRIBE TABLE`**
postgresql: **`\d+` table**

*** Query plans.** We use the EXPLAIN command to see the query plan used by PostgreSQL:

```
explain select * from authors where name = 'David J. DeWitt';

                                        QUERY PLAN
-----------------------------------------------------------------------------
Index Scan using authors_name on authors  (cost=0.43..8.45 rows=1 width=19)
Index Cond: ((name)::text = 'David J. DeWitt'::text)
```

This query plan uses an index scan based on authors.name. Its estimated costs are in units of disk page fetches, between 0.43 (time expended before output scan can start) and 8.45 (total cost). Furthermore, the output is expected to contain 1 answer of width 19 in bytes.

(1) Consider the following query:

```
select * from authors
where name < 'David J. DeWitt';
```

a) What is the query plan that PostgreSQL uses for this query? What is the textbook description of this type of query plan? (Or name the slide in the lecture notes that describe this type of query plan.)
b) What is the estimated number of rows? What is the actual number of rows returned by the query?
c) If the estimated number of rows differs from the actual number of rows returned, briefly explain how this happens. (Hint: Use pg_class and pg_stats to look for the statistics about the authors relation and the name attribute.)

Answer:

a) Query Plan:

```
explain select * from authors where name < 'David J. DeWitt';
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on authors  (cost=7918.93..24189.87 rows=321355 width=19)
   Recheck Cond: ((name)::text < 'David J. DeWitt'::text)
   ->  Bitmap Index Scan on authors_name  (cost=0.00..7838.59 rows=321355 width=0)
         Index Cond: ((name)::text < 'David J. DeWitt'::text)
 (4 rows)
```

Textbook description: Unclustered index + Sorting based on rid (Slide 8)

b) Actual number of rows:

```
select count(*) from authors where name < 'David J. DeWitt';

count
--------
 316440
(1 row)
```

Estimated number of rows:

```
rows=321355
```

c) Explanation for the difference:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'author%';

   relname    | relkind |  reltuples   | relpages
--------------+---------+-------------+----------
 authors      | r       | 1.69134e+06 |    12254
 authors_name | i       | 1.69134e+06 |     7142
 authors_pkey | i       | 1.69134e+06 |     4641
 (3 rows)


select tablename, attname, avg_width, histogram_bounds
from pg_stats
where tablename like 'authors%';

//results omitted due to the size of the histogram
```

The estimated number of query answers is `321355`. The number of returned tuples is `316440`.

The total number of tuples in the authors relational is `1691341`, or `1691340` when sampled from the index. There are 100 buckets in the histogram on <name>. The string `'David J. DeWitt'` is between the 20th boundary `"Daniel Fredholm"` and 21st boundary `"David Fleiszer"`. So the fraction of values less than `"Daniel Fredholm"` is estimated to be close to **19/100** of the total number of tuples, which turns out to be an **overestimate**.

(2) Consider the following query:

```
select * from authors
where name < 'A. A';
```

a) What is the query plan that PostgreSQL uses for this query? What is the textbook description of this type of query plan? (Or name the slide in the lecture notes that describe this type of query plan.)
b) Briefly explain why the plan chosen from PostgreSQL for this query differs from that for the previous query.

Answer:

a) Query Plan:

```
                                  QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using authors_name on authors  (cost=0.43..8.44 rows=1 width=19)
   Index Cond: ((name)::text < 'A. A'::text)
```

Textbook description: Unclustered index (Slide 8)

b) Now the index scan is **selective enough** so PostgreSQL decided to **follow the rids to fetch the tuples directly**, instead of using sorting to consolidate the page ids from all rids retrieved from the index.

(3) Consider the next query:

```
SELECT * FROM papers p, authors a, paperauths pa
WHERE pa.paperid = p.id
AND pa.authid = a.id
AND a.name = 'David J. DeWitt';
```

a) What is the query plan that PostgreSQL uses for this query?
b) Briefly explain why PostgreSQL chooses this plan.
c) What is the estimated number of returned rows? What is the actual number of rows returned by the query? Briefly explain why the estimated number of rows differs from the actual number of rows returned.

Answer:

a) Query Plan:

```
                                           QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop  (cost=5.63..184.81 rows=5 width=159)
   ->  Nested Loop  (cost=5.20..182.35 rows=5 width=27)
         ->  Index Scan using authors_name on authors a  (cost=0.43..8.45 rows=1 width=19)
               Index Cond: ((name)::text = 'David J. DeWitt'::text)
         ->  Bitmap Heap Scan on paperauths pa  (cost=4.77..173.47 rows=43 width=8)
               Recheck Cond: (authid = a.id)
               ->  Bitmap Index Scan on paperauths_pkey  (cost=0.00..4.76 rows=43 width=0)
                     Index Cond: (authid = a.id)
   ->  Index Scan using papers_pkey on papers p  (cost=0.43..0.48 rows=1 width=132)
         Index Cond: (id = pa.paperid)
 (10 rows)
```

b) PostgreSQL chooses this plan because (*i*) the predicate on **authors**.name is expected to return a single row, (*ii*) the join with **paperauths** can be facilitated using the index on < authid, paperid>, (iii) since the number of results of joining authors and paperauthors is expected to be small, again we can join these results with the **papers** using an index on the <id> attribute of the papers table.

c) Estimated number of returned rows is 5. The actual number of rows returned by the query is 186.

The estimated number of rows differs from the actual number of rows returned because the optimizer assumes that given a particular author, the number of matches in **paperauths** is the number of tuples in paperauths divided by the number of tuples in authors, that is,

```
8998448/1691340 = 5.3
```

which turned out to be a significant underestimate.

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'paperauth%';

     relname      | relkind |  reltuples   | relpages
------------------+---------+-------------+----------
 paperauths       | r       | 8.99845e+06 |    48641
 paperauths_pkey | i       | 8.99845e+06 |    24675
(2 rows)
```

If you wonder why the index scan on paperauths first estimates the number of rows to be 43, note that this estimate is for a single table index lookup (not part of a join). This estimate is the number of tuples divided by the number of distinct values of authid `8998448/210593` = `43`.

```
select tablename, attname, avg_width, n_distinct
from pg_stats
where tablename like 'paperaut%';

tablename  | attname | avg_width | n_distinct
------------+---------+-----------+------------
 paperauths | paperid |         4 |     ?
 paperauths | authid  |         4 |     210593
(2 rows)
```

(4) Now change the above query to:

```
SELECT * FROM papers p, authors a, paperauths pa
WHERE pa.paperid = p.id
AND pa.authid = a.id
AND a.name < 'David J. DeWitt';
```

a) What is the query plan that PostgreSQL uses for this query?
b) Briefly explain why PostgreSQL chooses this plan.

Answer:

a) Query Plan:

```
explain SELECT * FROM papers p, authors a, paperauths pa
WHERE pa.paperid = p.id
AND pa.authid = a.id
AND a.name < 'David J. DeWitt';

                                             QUERY PLAN
----------------------------------------------------------------------------------------------------
 Hash Join  (cost=229973.09..649323.77 rows=1709706 width=159)
   Hash Cond: (pa.paperid = p.id)
   ->  Hash Join  (cost=30089.80..336733.76 rows=1709706 width=27)
         Hash Cond: (pa.authid = a.id)
         ->  Seq Scan on paperauths pa  (cost=0.00..138625.48 rows=8998448 width=8)
         ->  Hash  (cost=24189.87..24189.87 rows=321355 width=19)
               ->  Bitmap Heap Scan on authors a  (cost=7918.93..24189.87 rows=321355 width=19)
                     Recheck Cond: ((name)::text < 'David J. DeWitt'::text)
                     ->  Bitmap Index Scan on authors_name  (cost=0.00..7838.59 rows=321355 width=0)
                           Index Cond: ((name)::text < 'David J. DeWitt'::text)
   ->  Hash  (cost=98944.57..98944.57 rows=3151257 width=132)
         ->  Seq Scan on papers p  (cost=0.00..98944.57 rows=3151257 width=132)
 (12 rows)
```

b) The plans are different because the estimate of the number of tuples matching the < x predicate changes with x. This causes Postgres to choose strategies appropriate for that number of tuples, **moving away from random index lookups as the number of matching tuples increases**, and so are the number of random seeks.

First, as the number of relevant authors is estimated to be `321355`, with the width of `19` bytes, totaling `321355 * 19 = 6105745`. These author tuples can be easily **fit in a hash table in memory**, so the query optimizer has chosen hash join, instead of the previous index nested loops join.

Second, the output of the first join, is estimated to be `1709706 * 27 = 46162062`. The optimizer decides to build a hash table over the other input, and pipeline the results of the first join to probe the in-memory hash table.