CS 586 Introduction to Databases
Assignment 7 – Query Evaluation, Transactions and Normalization
He, Haomin
11/28/2017

Question 1: Relational Algebra Equivalences
Each of the proposed equivalences below on relations r(R) and s(S) only holds under certain conditions. In the equivalences,
• X, R and S are sets of attributes,
• C is a single attribute,
• COUNT(r) returns the number of tuples in r, and
• ⋈ is natural join.

(a) (4 points each) For each equivalence, give example schemas and instances for r and s where the equivalence does not hold. (Assume that the expressions contain no syntax errors. For example, in ii, X is assumed to be a subset of R.)

i. $\pi_X (r \bowtie s) \equiv \pi_X (r) \bowtie s$
r(x, y) and s(y, z). If we natural join r and s, we get (x,y,z). When we project x, we get x column.
r(x, y) and s(y, z). If we project x from r first, we get x column. But from here we aren't able to natural join the x column with s(y, z). Because there is no common attribute to join them.

ii. $\pi_X (r - (\sigma_{C=5} (r))) \equiv \pi_X (r) - \pi_X (\sigma_{C=5} (r))$
Assume this is our table:

r table	c	x
	4	1
	5	1
	5	1

If we take difference between r and r(c=5), we get {(4, 1)} left. Then we project x, we get value 1.
If we select r(c=5) first, then project x, we get {(1)}. If we project x on r, we also get {(1)}. When we take difference between them, we get an empty set.

iii. COUNT(r ⋈ s) ≡ COUNT(r) * COUNT(s)
Assume these are out tables:

s table	c	x
	1	d
	1	e
	2	f
	2	g
	3	h

r table	c	y
	1	a
	2	b
	3	c

join table	c	x	y
	1	d	a
	1	e	a
	2	f	b
	2	g	b
	3	h	c

We natural join r and s table, and count the rows, we get 5 rows.
However, COUNT(r) * COUNT(s) = 3 * 5 = 15 rows.

iv. COUNT(r ⋈ s) ≡ COUNT(r)
Let's use the tables from the (iii).
We natural join r and s table, and count the rows, we get 5 rows.
However, COUNT(r) = 3 rows.

(b) (6 points) For each equivalence, give a side condition that guarantees the equivalence will hold. The condition should be at the schema level (relation schemes, keys, foreign keys) and not at the instance level (number of tuples, specific values). The less restrictive the condition, the better.

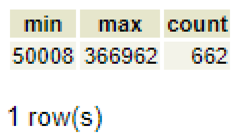
i. π_x (r ⋈ s) ≡ π_x (r) ⋈ s
Side condition: the join condition is based on the project attribute, in other words, we project x column, and natural join with r.x = s.x. Also in s table, there is only one column, which is x column.

ii. π_x (r – ($\sigma_{C=5}$ (r))) ≡ π_x (r) – π_x ($\sigma_{C=5}$ (r))
Side condition: the project attribute must be uniquely identified, in other words, there is no duplicate values in x column.

iii. COUNT(r ⋈ s) ≡ COUNT(r) * COUNT(s)
Side condition: there is only one row in either r table or in s table. And in this row, there is a valid natural join value that can be used for joining with the other table.

iv. COUNT(r ⋈ s) ≡ COUNT(r)
Side condition: r table ($\underline{x}$, c) where x is primary key, c is a foreign key refers to s table.
And s table ($\underline{c}$, y) where c is primary key. So that we can get COUNT(r ⋈ s) ≡ COUNT(r).

Question 2: Statistics
a. (5 points) Determine the min and max salary values in the agent table, and the number of rows in that table.
SELECT min(salary), max(salary), count(*) FROM spy.agent;



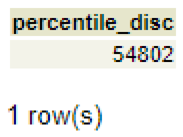
```
 min    max    count
50008  366962   662

1 row(s)
```

b. (5 points) Give an estimate for the number of rows in agent with salary < 75000, assuming a uniform distribution of salaries between min and max salary. Explain how you derived your estimate.

1/(366962 - 50008) = 0.00000315503
Probability = P(50008 < salary < 75000) = (75000 - 50008) * 0.00000315503 = 0.07885050976
Estimate for the number of rows = 662 * 0.07885050976 = 52.1990374611
So approximately 53 agents have salary < 75000.

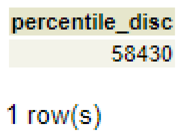
c. (5 points) Find the 25th, 50th and 75th percentile values for salaries in the agent table. (The 50th percentile value, for instance, is the smallest number s such that 50% of the rows have salary value less than or equal to s.)

```
SELECT percentile_disc(0.25)
WITHIN GROUP (ORDER BY salary ASC)
FROM spy.agent
;
```



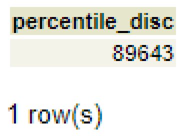
```
percentile_disc
          54802

1 row(s)
```

```
SELECT percentile_disc(0.50)
WITHIN GROUP (ORDER BY salary ASC)
FROM spy.agent
;
```



```
percentile_disc
          58430

1 row(s)
```

```
SELECT percentile_disc(0.75)
WITHIN GROUP (ORDER BY salary ASC)
FROM spy.agent
;
```



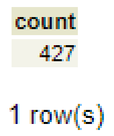
```
percentile_disc
          89643

1 row(s)
```

d. (5 points) Give an estimate of the number of rows in agent with salary < 75000, assuming in a uniform distribution in each quartile determined in c. Explain how you derived your estimate.
Because 58430 < 75000 < 89643 and 50% < 75000 < 75%
(75000 - 58430) / (89643 - 58430) = 0.53086854836

0.53086854836 * (75% - 50%) + 50% = 0.13271713709 + 0.5 = 0.63271713709
Estimate for the number of rows = 662 * 0.63271713709 = 418.858744754
So approximately 419 agents have salary < 75000.

e. (5 points) How many rows in agent actually have salary < 75000?

```
SELECT count(*) FROM spy.agent
WHERE salary < 75000
;
```



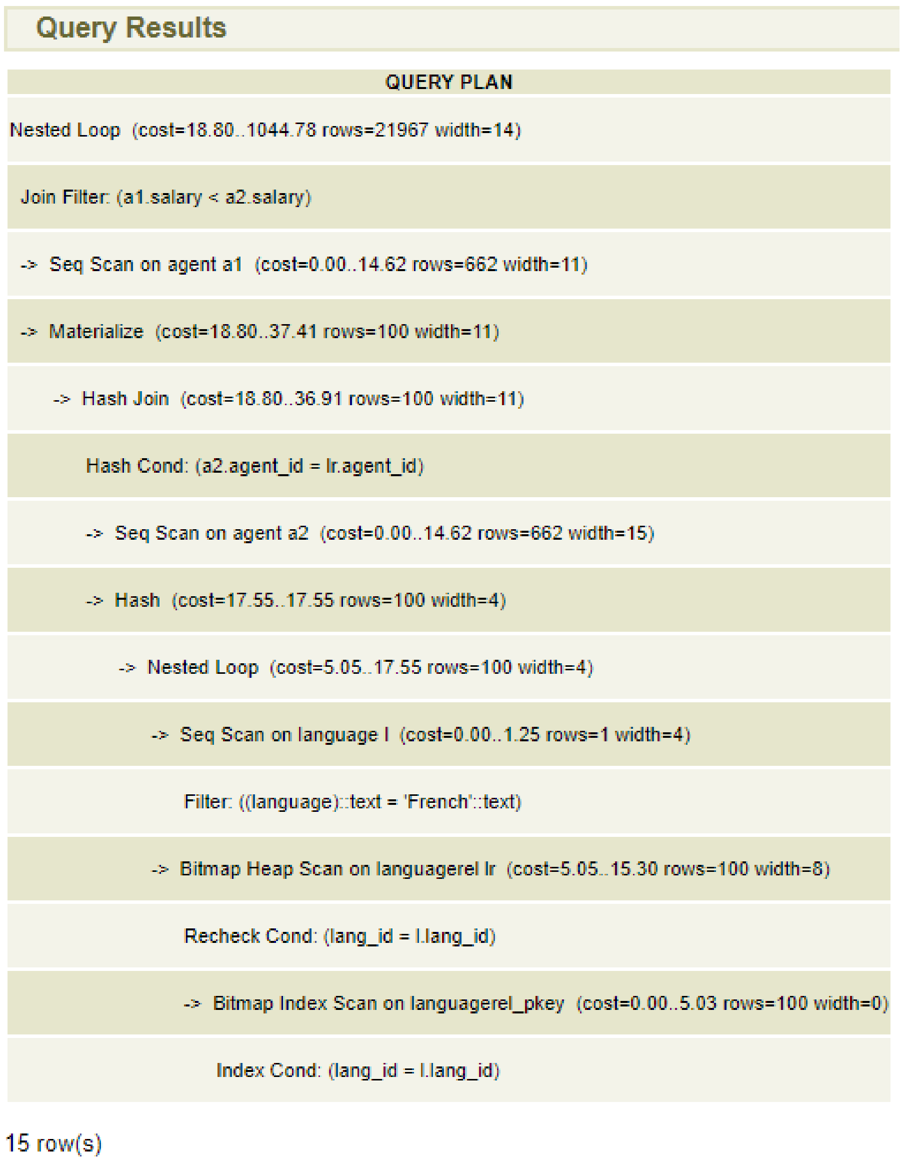
```
count
  427

1 row(s)
```

Question 3: Query Plans

For each SQL statement below, draw the query plan that Postgres chooses (which you can obtain with the EXPLAIN command). For each plan, suggest a reason that the particular join algorithms were chosen.

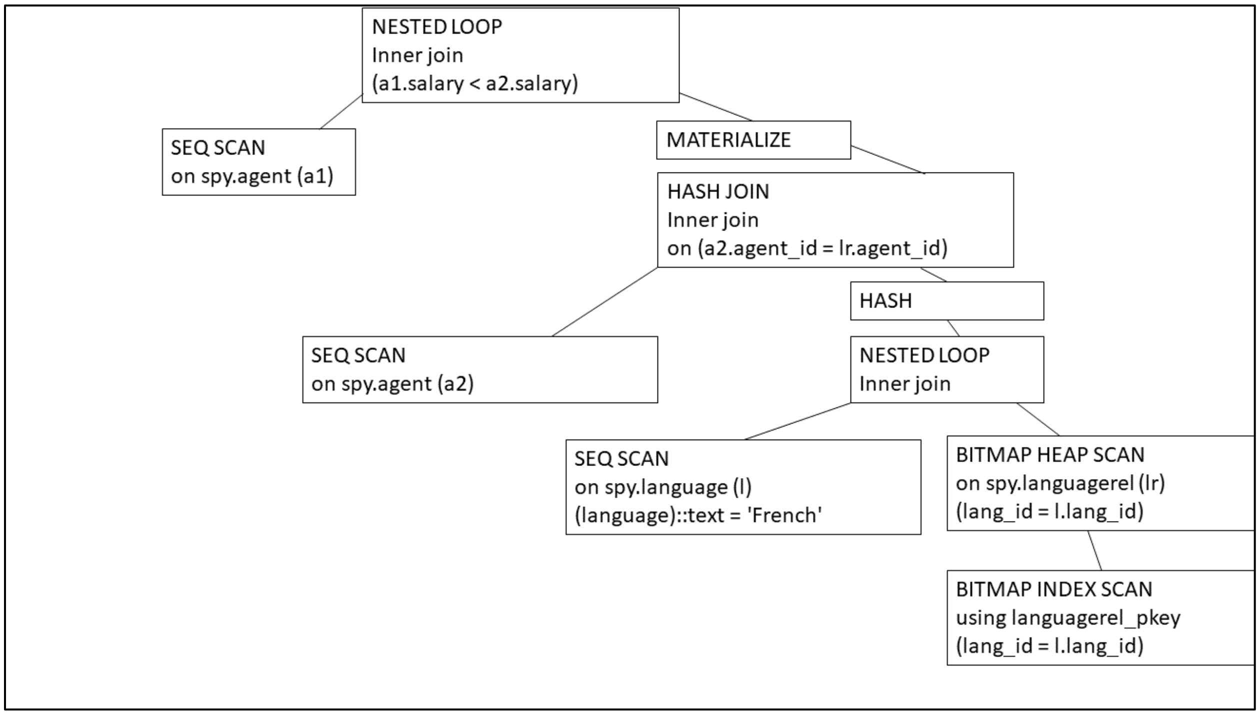
a. (10 points)

```
SELECT A1.last, A2.last
FROM agent A1, agent A2, languagerel LR, Language L
WHERE A1.salary < A2.salary AND A2.agent_ID = LR.agent_ID
AND LR.lang_ID = L.lang_ID AND L.language = 'French';
```


Query Results

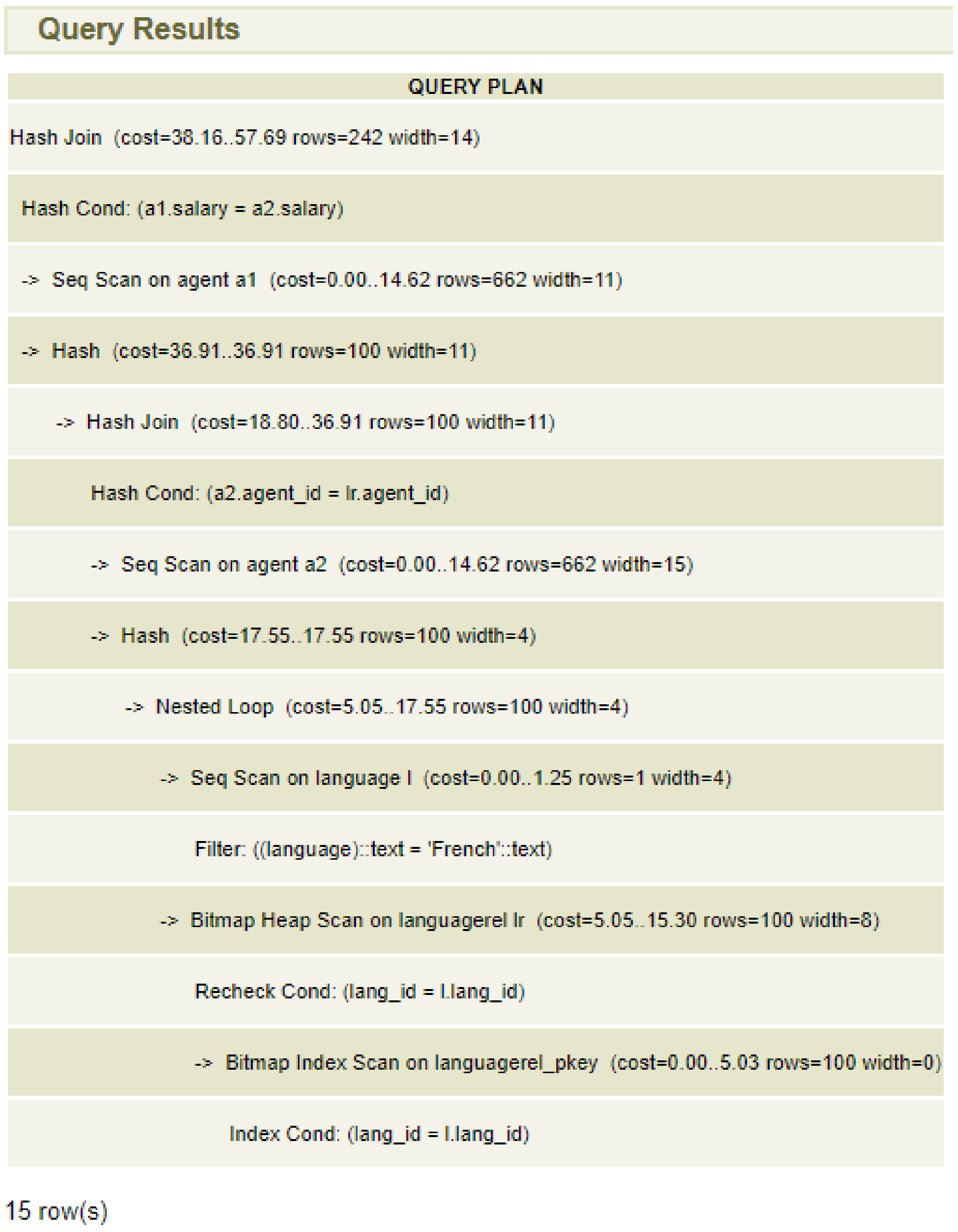
```
QUERY PLAN
Nested Loop  (cost=18.80..1044.78 rows=21967 width=14)
  Join Filter: (a1.salary < a2.salary)
  ->  Seq Scan on agent a1  (cost=0.00..14.62 rows=662 width=11)
  ->  Materialize  (cost=18.80..37.41 rows=100 width=11)
        ->  Hash Join  (cost=18.80..36.91 rows=100 width=11)
              Hash Cond: (a2.agent_id = lr.agent_id)
              ->  Seq Scan on agent a2  (cost=0.00..14.62 rows=662 width=15)
              ->  Hash  (cost=17.55..17.55 rows=100 width=4)
                    ->  Nested Loop  (cost=5.05..17.55 rows=100 width=4)
                          ->  Seq Scan on language l  (cost=0.00..1.25 rows=1 width=4)
                                Filter: ((language)::text = 'French'::text)
                          ->  Bitmap Heap Scan on languagerel lr  (cost=5.05..15.30 rows=100 width=8)
                                Recheck Cond: (lang_id = l.lang_id)
                                ->  Bitmap Index Scan on languagerel_pkey  (cost=0.00..5.03 rows=100 width=0)
                                      Index Cond: (lang_id = l.lang_id)
15 row(s)
```

Nested Loops Join: The most straightforward algorithm is Nested Loops Join. For each row on the left-hand side, the right-hand side is scanned for a match based on the join condition. So that we can find all rows that satisfy A1.salary < A2.salary, L.language = 'French' conditions. This bottom up method can get rid of lots of rows that are not French.
Hash Join: A Hash Join is the most versatile join method supported by the SQL Anywhere database server. The Hash Join algorithm builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches. So that we can find all rows that satisfy a2.agent_id = lr.agent_id condition.
Bitmap Heap Scan and Bitmap Index Scan: Then we use these two scans to find all rows that satisfy LR.lang_ID = L.lang_ID condition.



b. (10 points)

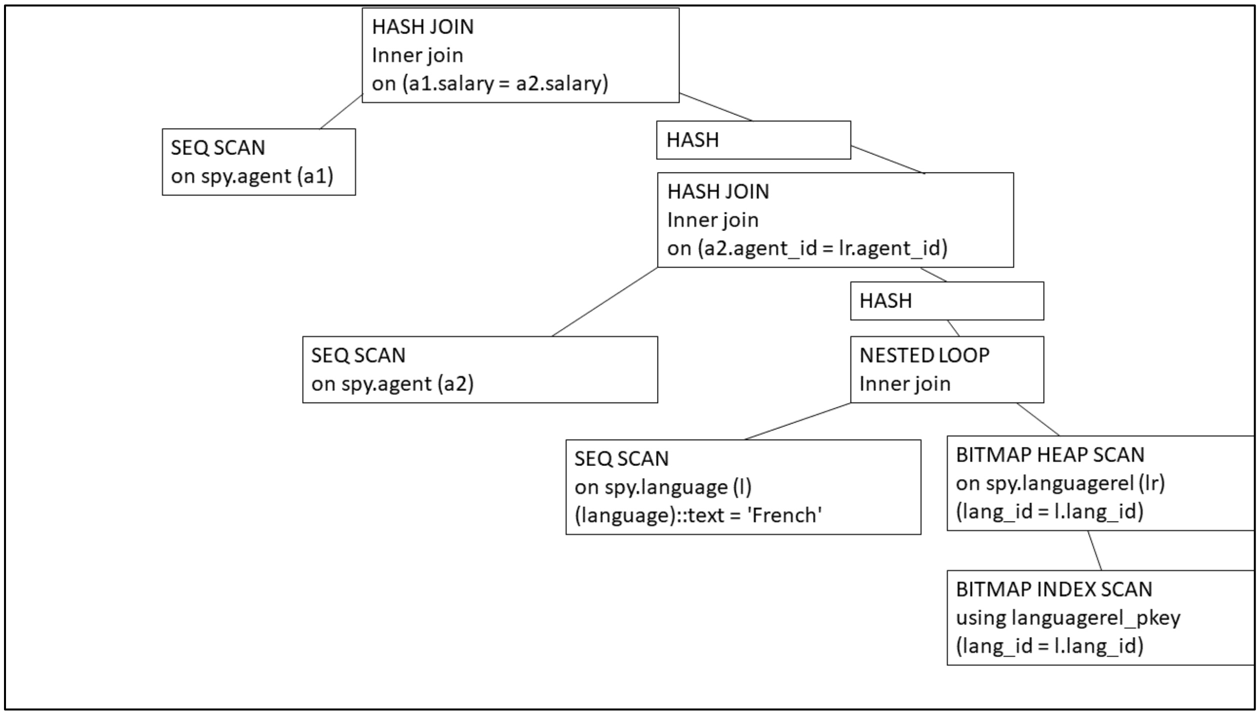
```
SELECT A1.last, A2.last
FROM agent A1, agent A2, languagerel LR, Language L
WHERE A1.salary = A2.salary AND A2.agent_ID = LR.agent_ID
AND LR.lang_ID = L.lang_ID AND L.language = 'French';
```


Query Results

```
QUERY PLAN
Hash Join  (cost=38.16..57.69 rows=242 width=14)
  Hash Cond: (a1.salary = a2.salary)
  ->  Seq Scan on agent a1  (cost=0.00..14.62 rows=662 width=11)
  ->  Hash  (cost=36.91..36.91 rows=100 width=11)
        ->  Hash Join  (cost=18.80..36.91 rows=100 width=11)
              Hash Cond: (a2.agent_id = lr.agent_id)
              ->  Seq Scan on agent a2  (cost=0.00..14.62 rows=662 width=15)
              ->  Hash  (cost=17.55..17.55 rows=100 width=4)
                    ->  Nested Loop  (cost=5.05..17.55 rows=100 width=4)
                          ->  Seq Scan on language l  (cost=0.00..1.25 rows=1 width=4)
                                Filter: ((language)::text = 'French'::text)
                          ->  Bitmap Heap Scan on languagerel lr  (cost=5.05..15.30 rows=100 width=8)
                                Recheck Cond: (lang_id = l.lang_id)
                                ->  Bitmap Index Scan on languagerel_pkey  (cost=0.00..5.03 rows=100 width=0)
                                      Index Cond: (lang_id = l.lang_id)
```

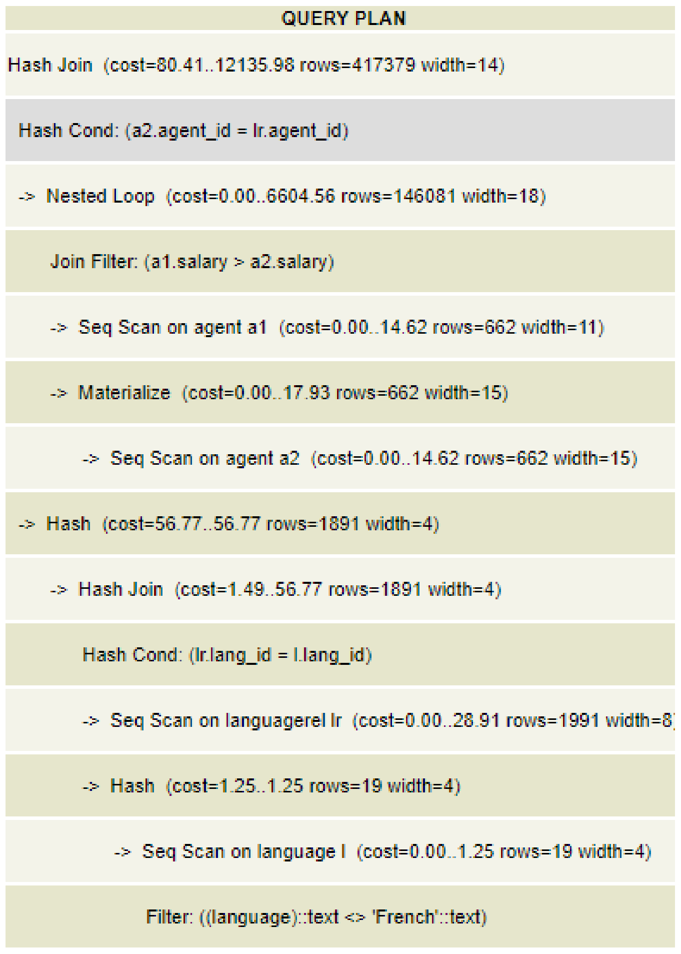
15 row(s)

Hash Join: A Hash Join is the most versatile join method supported by the SQL Anywhere database server. The Hash Join algorithm builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches. So that we can find all rows that satisfy a1.salary = a2.salary, a2.agent_id = lr.agent_id conditions.
Nested Loops Join: The most straightforward algorithm is Nested Loops Join. For each row on the left-hand side, the right-hand side is scanned for a match based on the join condition. So that we can find all rows that satisfy L.language = 'French' condition. This bottom up method can get rid of lots of rows that are not French.
Bitmap Heap Scan and Bitmap Index Scan: Then we use these two scans to find all rows that satisfy LR.lang_ID = L.lang_ID condition.



c. (10 points)

```
SELECT A1.last, A2.last
FROM agent A1, agent A2, languagerel LR, Language L
WHERE A1.salary > A2.salary AND A2.agent_ID = LR.agent_ID
AND LR.lang_ID = L.lang_ID AND L.language <> 'French';
```

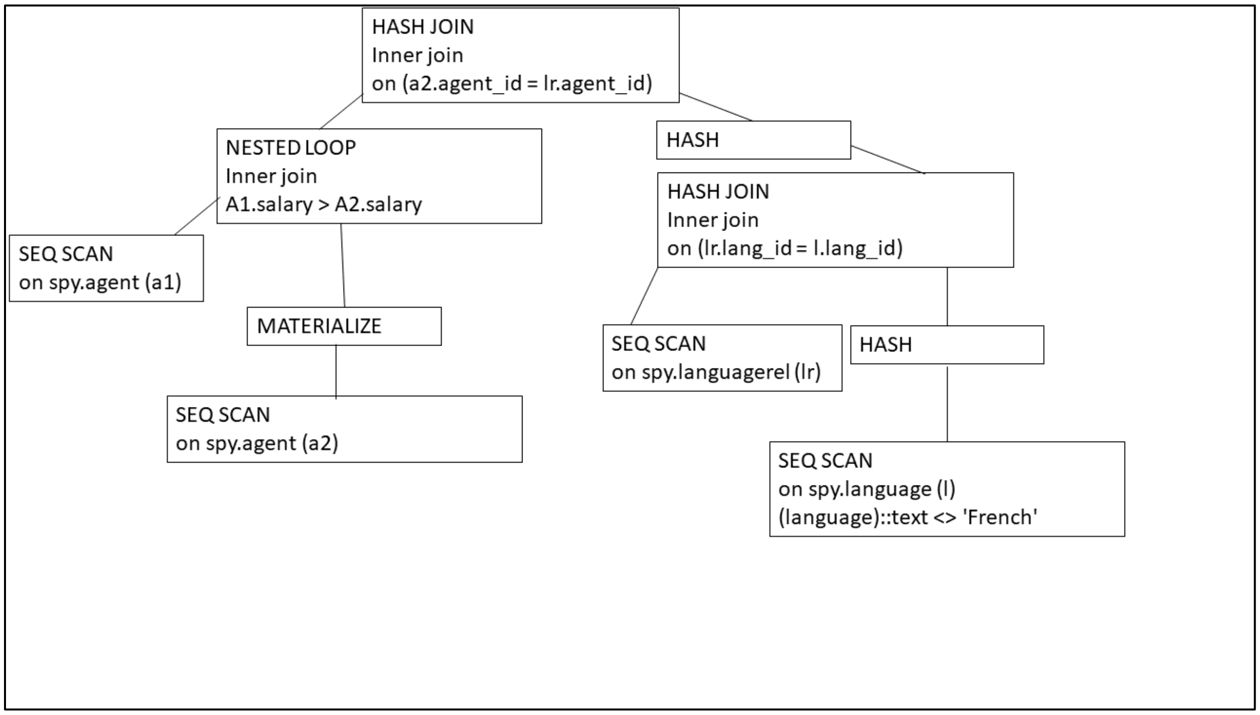


```
QUERY PLAN
Hash Join  (cost=80.41..12135.98 rows=417379 width=14)
  Hash Cond: (a2.agent_id = lr.agent_id)
  ->  Nested Loop  (cost=0.00..6604.56 rows=146081 width=18)
        Join Filter: (a1.salary > a2.salary)
        ->  Seq Scan on agent a1  (cost=0.00..14.62 rows=662 width=11)
        ->  Materialize  (cost=0.00..17.93 rows=662 width=15)
              ->  Seq Scan on agent a2  (cost=0.00..14.62 rows=662 width=15)
  ->  Hash  (cost=56.77..56.77 rows=1891 width=4)
        ->  Hash Join  (cost=1.49..56.77 rows=1891 width=4)
              Hash Cond: (lr.lang_id = l.lang_id)
              ->  Seq Scan on languagerel lr  (cost=0.00..28.91 rows=1991 width=8
              ->  Hash  (cost=1.25..1.25 rows=19 width=4)
                    ->  Seq Scan on language l  (cost=0.00..1.25 rows=19 width=4)
                          Filter: ((language)::text <> 'French'::text)
```

14 row(s)

Hash Join: A Hash Join is the most versatile join method supported by the SQL Anywhere database server. The Hash Join algorithm builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches. So that we can find all rows that satisfy a2.agent_id = lr.agent_id, lr.lang_id = l.lang_id, L.language <> 'French' conditions. This bottom up method can get rid of lots of rows that are French.

Nested Loops Join: The most straightforward algorithm is Nested Loops Join. For each row on the left-hand side, the right-hand side is scanned for a match based on the join condition. So that we can find all rows that satisfy a1.salary > a2.salary condition.



Question 4: Recovery (10 points)
There have been proposals to use multiple log files for recovery. Log records can be appended to any one of the files, but all records from the same transaction must go to the same file. Give one advantage of multiple log files and one disadvantage.
Advantage: increasing the number of transaction log files for a given database will improve the write performance to the database.
Disadvantage: having multiple log files implies that the first one ran out of space and because the second one still exists, the first one might still be large (maybe the second one is large too). During disaster recovery. If the log files don't exist, they must be created and zero-initialized, and twice if you restore a different backup too as both the full and different restores zero out the log. If the first log file is as big as it can be, and there's a second log file still, that's potentially a lot of log file to zero initialize, which translates into downtime during disaster recovery.

Question 5: Decomposition
Consider the following relational schema:
viols(VID VT VD InD BID SNu SNa SCo Bro BrI Zip).

a. (15 points) Show a decomposition of violations into BCNF. Show each step in your decomposition and the keys for each relation scheme. (You can have more than one key per scheme.)
Because VID -> VT VD InD BID SNu SNa SCo Bro BrI Zip
And BID -> SNu SNa SCo Bro BrI Zip
V1(<u>VID</u> VT VD InD BID)

V2(BID SNu SNa SCo Bro BrI Zip)

Because Bro -> BrI
And BrI -> Bro
V1(VID VT VD InD BID)
V2(BID SNu SNa SCo Bro Zip)
V3(Bro BrI)

Because VT -> VD
And VD -> VT
V1(VID VT InD BID)
V2(BID SNu SNa SCo Bro Zip)
V3(Bro BrI)
V4(VT VD)

Because SCo -> SNa
V1(VID VT InD BID)
V2(BID SNu SCo Bro Zip)
V3(Bro BrI)
V4(VT VD)
V5(SCo SNa)

Because SNu SCo Zip -> Bro
V1(VID VT InD BID)
V2(BID SNu SCo Bro Zip)
V3(Bro BrI)
V4(VT VD)
V5(SCo SNa)

b. (5 points) Find a functional dependency in the original scheme that does not correspond to a key in your normalized scheme.
Answer: SNu SCo Zip -> Bro

c. (10 points) Is it possible that a legal (satisfies all keys) database instance d on your normalized scheme violates the FD in b., in the sense that if you join all the relations in d together, the result r violates the FD? Explain why or why not.
Answer: The decomposition algorithm (based on lifting "troublesome" FDs into a separate table) guarantees that the decomposition of the original table is lossless. However, if we throw away information from the original schema, the decomposition is lossy, and join all the relations in d would give you too many tuples.

As long as, the attributes in common are a key for (at least) one of the relations, then we know that the decomposition is lossless.
We must follow the decomposition steps:
1. Lift the "troublesome" FD(s) (all the FDs with the same LHS) into a table of their own. Key for new table is left hand side of
the troublesome FD(s).
2. Leave the left side of the FD behind in the original table.
3. Eliminate the RHS attributes from the original table.

No, it does not violate the FD.
If we take the functional dependency SNu SCo Zip -> Bro out of the normalized scheme, we would not lose information of BID -> SNu SNa SCo Bro BrI Zip. And we know BID -> Bro -> BrI can still be satisfied.
V1(VID VT InD BID)
V2(BID SNu SCo Zip)
V3(Bro BrI)
V4(VT VD)
V5(SCo SNa)
V6(SNu SCo Zip Bro)

References:
Lecture slides
http://dcx.sybase.com/1200/en/dbusage/join-methods-optimizer-queryopt.html
http://tatiyants.com/pev/#/plans
https://dba.stackexchange.com/questions/62344/multiple-transaction-log-files-and-performance-impact
https://www.sqlskills.com/blogs/paul/multiple-log-files-and-why-theyre-bad/