CSE414 Final Exam
Spring 2018
June 7, 2018

- Please read all instructions (including these) carefully.
- **This is a <u>closed-book exam</u>. You are allowed two pages of note sheets that you can write on both sides.**
- Write your name and UW student number below.
- No electronic devices are allowed, including **cell phones** used merely as watches. Silence your cell phones and place them in your bag.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are 20 pages in this exam, including this one.
- There are 8 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 110 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the blank pages as scratch paper. **Do not** use any additional scratch paper.
- Relax. You are here to learn. Good luck!

By writing your name below, you certify that you have not received any unpermitted aid for this exam, and that you will not disclose the contents of the exam to anyone in the class who has not taken it.

NAME: ______Solutions_______________ SECTION: ____________

STUDENT NUMBER: _________________________________

Text in red is answer explanation or clarifications made during the exam.

Problem	**Points**	**Problem**	**Points**
1	20	5	15
2	45	6	34
3	34	7	25
4	25	8	2
Total		**200**	

Problem 1: Warm Up (20 points total)

Select either True or False for each of the following questions. For each question you get 2 points for answering it correctly, -1 point for an incorrect answer, and 0 point for no answer. The minimum you will get for this entire problem is 0.

a) 2PL can be used to enforce ACID guarantees.

True [✓ **False**]

b) $\sigma_{a>10}(R) \bowtie S$ always has lower cost than $\sigma_{a>10}(R \bowtie S)$

True [✓ **False**]

c) Conflict serializability ensures that the phantom problem does not exist.

True [✓ **False**]

d) BCNF does not guarantee that all functional dependencies can be checked after decomposition.

[✓ **True**] False

e) The shuffling operation in MapReduce can be implemented using relational algebra.

[✓ **True**] False

f) There can only be one dense index for a given relation.

[✓ **True**] False

g) Executing $R \bowtie S$ using broadcast join is the most efficient when S has many more tuples than R and is broadcasted.

True [✓ **False**]

h) In DBMS with multiple lock modes, all reads must precede by acquiring an exclusive lock.

True [✓ **False**]

i) Json documents do not need to be in first normal form.

[✓ **True**] False

j) All natural joins are also theta joins.

Natural joins remove duplicates, but theta joins do not.

True [✓ **False**]

Problem 2: Transactions (45 points total)

a) For each of the following schedules, indicate which kind of schedule it is. Circle all that applies. Here $\texttt{Co}_i$ means transaction i commits, and $\texttt{Ab}_i$ means transaction i aborts. For each circle you get 4 points if it is correct, and -2 points if it isn't. The minimum you will get for this entire problem is 0. (8 points total)

i) $R_1(X);\ W_2(X);\ W_1(X);\ R_3(X);\ Co_1;\ Co_2;\ Co_3$

Serial Serializable Conflict-serializable [✓ None of the above]

ii) $W_1(X);\ R_2(X);\ W_1(X);\ Co_2;\ Ab_1$

Serial [✓ Serializable] Conflict-serializable None of the above

Serializable to T_2; T_1 since T_1 is aborted

b) Is it possible for a serializable schedule to be not conflict-serializable? If not, write "NO" below. Otherwise, give an example of such a schedule. If your schedule involves writes, then also indicate the actual value that was written. For instance, $W_1(X,10)$ means that transaction 1 writes 10 to data element X. (4 points)

Yes. For instance the schedule $W_1(X, 10); W_2(X, 10); R_1(X)$ is serializable to $T_1 \rightarrow T_2$, but is not conflict-serializable.

c) For each schedule below, determine if the associated statement is true or not. Again $\texttt{Co}_i$ means transaction i commits. Assume there is only one type of lock for lock-based protocols, and that all transactions are to be executed under serializable isolation level. You get 2 points for the correct answer, -1 point for the incorrect one, and 0 for no answer. The minimum you will get for this entire problem is 0. (12 points total)

i) $R_1(A);\ R_2(A);\ W_2(B);\ W_1(A);\ R_1(C);\ Co_1;\ Co_2;$

This schedule is serializable.	**True**	False
This schedule can result from the two-phase locking protocol.	**True**	False
This schedule can result from the strict two-phase locking protocol.	True	**False**

ii) $R_1(A);\ R_3(A);\ W_2(B);\ Co_2;\ W_3(C);\ W_1(B);\ Co_1;\ Co_3;$

This schedule is serializable.	**True**	False
This schedule can result from the two-phase locking protocol.	**True**	False
This schedule can result from the strict two-phase locking protocol.	**True**	False

d) After TAing 414, Cindy has grown tired of locking and invents a new transactional protocol based on **operation timestamps**. In this protocol, the DBMS maintains an append-only **transaction log**. Rather than using locks, each transaction T proceeds as follows:

- Start: append `START(T,ts)` to the log, where `ts` is the timestamp when T started.
- Reads: read values from the disk directly, and append `READ(T,e,ts)` to the log, where `e` is the data element that is read, and `ts` is the timestamp of the read.
- Writes: append `WRITE(T,e,ts)` to the log, where `e` is the data element to write, and `ts` is the timestamp of the write. **Do not actually write to the disk.**
- Commit: call `check`, a function provided by the DBMS. `check(T)` will either abort the transaction by ignoring T's writes and appending `ABORT(T,ts)` to the log, or commit by propagating the writes described in the log to the disk and appending `COMMIT(T,ts)` to the log. You can assume that all writes are propagated instantaneously to the disk should the DBMS decides to commit the transaction.

There is no lock involved in any of the operations above! Your job is to help Cindy determine when should `check` commits or aborts a transaction.

Suppose we want to achieve the **serializable** isolation level, and our goal is to commit as many transactions as allowable (i.e., aborting everything is not an option). Given the following logs, determine whether the associated statement is true or not. You get 2 points for the correct answer, -1 point for the incorrect one, and 0 for no answer. The minimum you will get for this entire problem is 0. (8 points total for (i) to (iii))

i) START(T_1,0); START(T_2,1); READ(T_2,X,2); WRITE(T_1,X,3); COMMIT(T_1,4);

What should check(T_2) return? Commit [✓ Abort]

ii) START(T_1,0); START(T_2,1); WRITE(T_1,X,2); READ(T_2,X,3); ABORT(T_1,4);

What should check(T_2) return? [✓ Commit] Abort

iii) START(T_1,0); START(T_2,1); READ(T_2,X,2); WRITE(T_1,Y,3); START(T_3,4); WRITE(T_3,Y,5); COMMIT(T_3,6);

What should check(T_1) return? [✓ Commit] Abort

What should check(T_2) return? [✓ Commit] Abort

iv) Describe in a few sentences the general algorithm for determining when `check(T)` should return `COMMIT` or `ABORT` for transaction T, in terms of the writes performed by T (i.e., `WS(T)`), and the reads performed by T (i.e., `RS(T)`). (5 points)

Check that RS(T) ∩ WS(U) and WS(T) ∩ WS(U) are both empty, for any transaction U that has started but not yet committed or aborted. If true then return COMMIT for T, otherwise return ABORT.

v) Describe what type of transaction workloads, if any, will benefit from Cindy's protocol, as compared to the lock-based protocols discussed in class. (e.g., "When all transactions reads the same data element." [not the answer]) (4 points)

Many answers are possible, e.g., when transactions are read-only / no conflicts among the transactions.

vi) Describe what type of transaction workloads, if any, will benefit from lock-based protocols discussed in class, as compared to Cindy's protocol. (4 points)

Many answers are possible, e.g., when transactions have a lot of conflicts among them.

Problem 3: Queries (34 points total)

Jonathan is looking for a new job, but is very particular about what he wants from the companies.

- It has to be a big company in Seattle that doesn't pay terribly (3a)
- A large portion of the company budget should go to engineers (3b)
- The company shouldn't have too many or too few DBA's that only worked for one company. (3c)

Can you write queries to help him find his perfect company based on the following tables?

```
Company (cid, name, inSeattle, budget)
-- inSeattle = 1 if company is in Seattle, 0 otherwise.
Engineer (eid, specialization, salary)
WorksFor (eid, cid) -- eid is FK to Engineer, cid is FK to Company
```

a) Write a SQL query that returns the cid's of Seattle companies that hire more than 1000 engineers, with none of the hired engineers earning the lowest salary among all engineers.

Write this query *without* using `WITH` or subqueries for full credit. Name the resulting column cid. (8 points) Hint: `COUNT(DISTINCT X)` gives a count of unique entries for attribute X.

```
SELECT C.cid
FROM Company AS C, WorksFor AS W, Engineer E1, Engineer E2
WHERE C.inSeattle = 1
                  AND C.cid = W.cid
                  AND W.eid = E1.eid
                  AND E1.salary > E2.salary
GROUP BY C.cid
HAVING COUNT(DISTINCT W.eid) > 1000;
```

b) Write a SQL query that returns the cid of the company(s) with the highest total salary to budget ratio of all companies. Name the resulting column cid. (8 points)

```
WITH temp AS (
SELECT C.cid AS id, SUM(E.salary)/C.budget as ratio
FROM Company AS C, WorksFor AS W, Engineer AS E
WHERE C.cid = W.cid AND W.eid = E.rid
GROUP BY C.cid, C.budget)

SELECT T.cid
FROM temp T
WHERE T.ratio >= ALL (SELECT T2.ratio FROM temp T2)
```

c) Write a SQL query that returns the cid of the company(s) that, of all companies, have neither the most nor the least number of engineers who specialize in "DBA", and all those engineers work at a single company only. Name the resulting column cid. (8 points)

```
WITH temp AS (
SELECT W1.cid AS id, COUNT(*) AS count
FROM WorksFor AS W1, Engineer AS E
WHERE W1.eid = E.eid AND E.specialization = 'DBA'
AND NOT EXISTS (SELECT *
                FROM WorksFor W2
                WHERE W2.eid = W1.eid AND W2.cid <> W1.cid)
                GROUP BY W1.cid)

SELECT T.id as cid
FROM temp AS T
WHERE T.count < ANY (SELECT T2.count FROM temp AS T2)
       AND T.count > ANY (SELECT T3.count FROM temp AS T3)
```

Outer query can also be a three way join and select distinct instead of ANY

d) Describe in one sentence how Jonathan can find the name of the companies to which he should apply by utilizing the results from a-c. (2 points)

Take the intersection of the three queries and join with the Company table to find the names.

e) Write a safe Datalog query that returns the eid of all engineers that only work at companies that hired more than 414 engineers, with each of those engineers making the lowest salary among all engineers. (8 points)

```
Ans(x) :- !nonAns(x), Engineer(x,_,_)

nonAns(x) :- WorksFor(x,y), z=count { !NotLowest(e), WorksFor(e,y) },
             z <= 414

NotLowest(e) :- Engineer(e,_,s1), Engineer(f,_,_s2), s1>s2
```

NotLowest: There exists an engineer that makes less than e

nonAns: Engineer x, worked at a company that had less than or equal to 414 lowest paid engineers

Ans: Engineer x, did not work at a company that had less than or equal to 414 lowest paid engineers

Problem 4: Conceptual Design (25 points total)

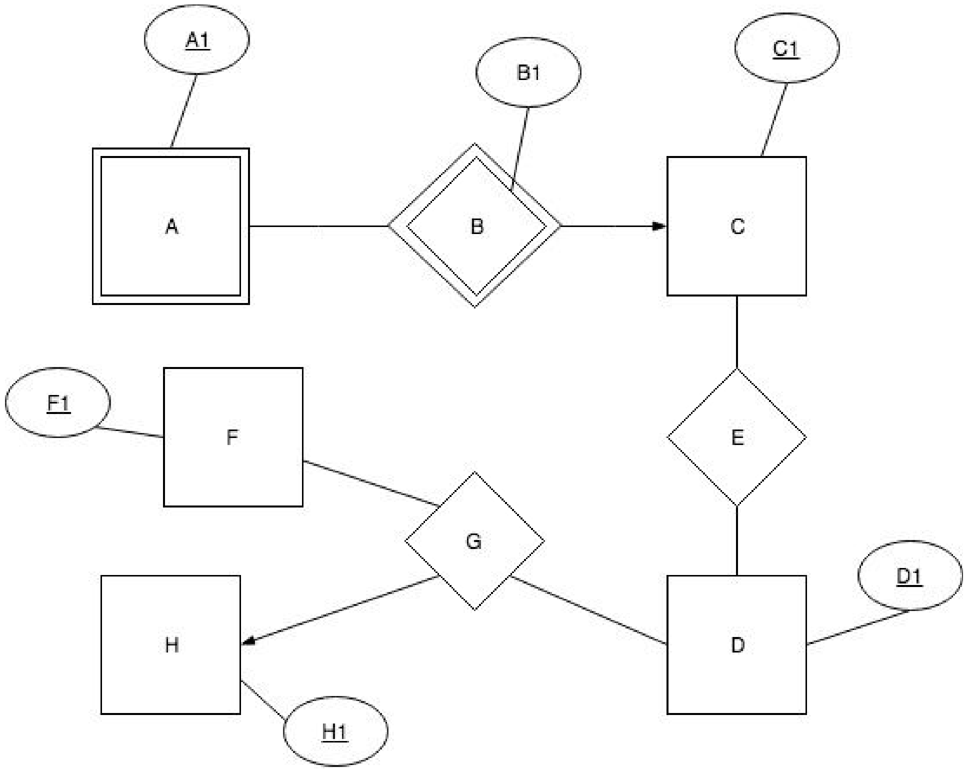
a) Given `R(A, B, C, D, E, F)` and the following functional dependencies:

- $B \rightarrow A, C$
- $C \rightarrow D$
- $F \rightarrow E$

Normalize R to BCNF and indicate the key in each relation. You just need to write down the final decomposition, no need to show steps. (10 points)

```
R1(A, B, C), R2(C, D), R3(B, F), R4(E, F)
```

b) Represent the E/R diagram below in relations. Make sure you indicate which attributes are the keys in each relation. (15 points)



```
A(A1, B1, C1)
C(C1)
E(C1, D1)
D(D1)
F(F1)
H(H1)
G(F1, D1, H1)
```

Problem 5: Parallel Query Processing (15 points total)

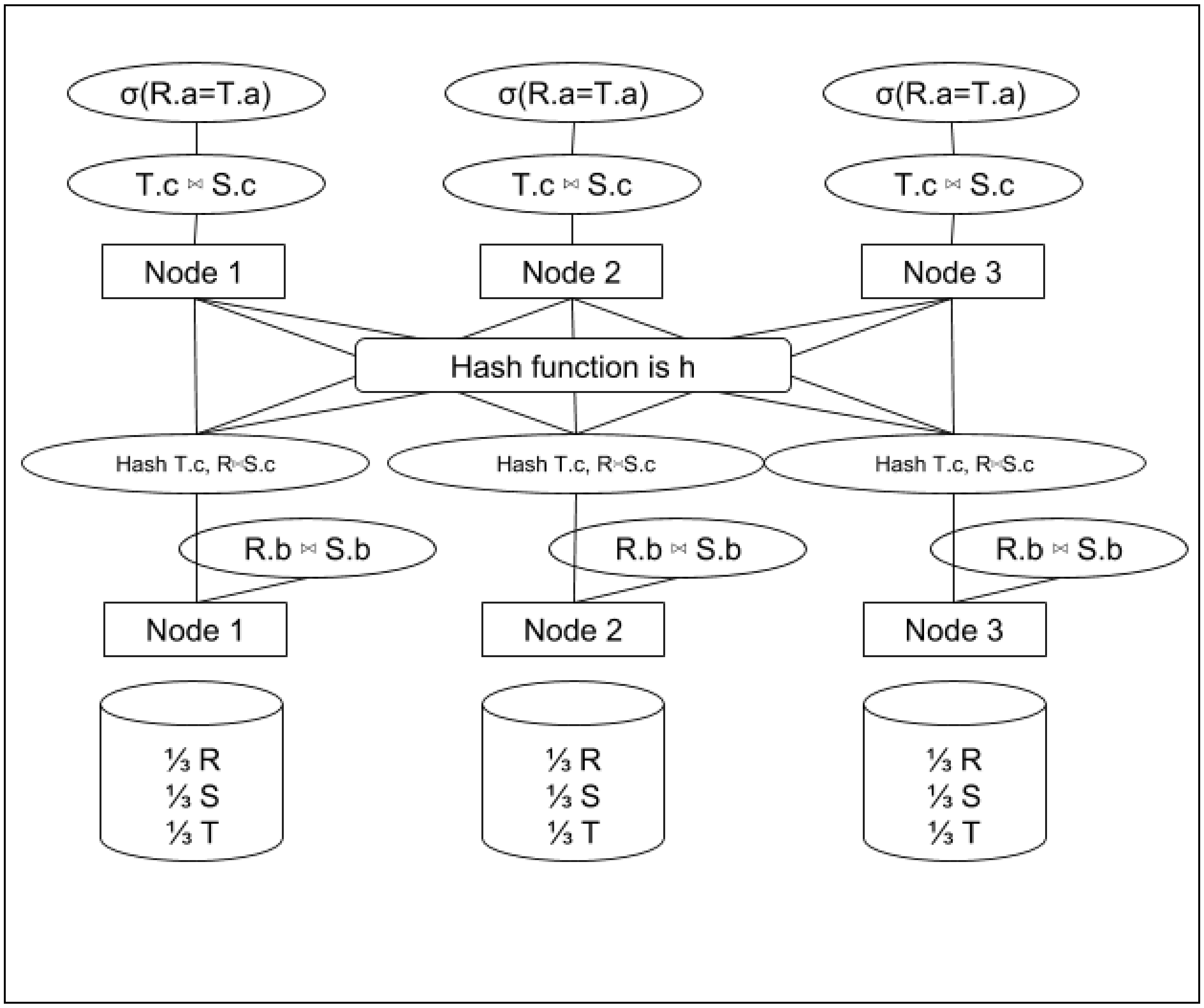
We have three relations `R(a,b)`, `S(b,c)`, `T(c,a)` and would like to compute the query below in parallel across 3 nodes:

```
SELECT *
FROM R, S, T
WHERE R.b = S.b AND S.c = T.c AND T.a = R.a;
```

`R` is hash-partitioned on `R.b` via the function `h`. `S` is hash-partitioned on `S.b` via the function `h`. `T` is block-partitioned. Draw a parallel query plan that will compute the query and specify any hash functions you use.

Errata: Hint is somewhat misleading. The solution technically requires 2 shuffles but they can be done at the same time.

You can assume tuples are evenly distributed. (Hint: you only need to shuffle once)



Problem 6: Query Optimization (34 points total)

We have the following database of UW employees and their main projects:

```
Employee(SSN, yearBorn, fname, lname)
Project(PID, SSN, yearStarted, pname, budget) -- SSN is FK to Employee
```

And we have the following meta-information:

Employee	Project
T(Employee) = 200,000	T(Project) = 200,000
B(Employee) = 10,000	B(Project) = 10,000
V(Employee, fname) = 200	V(Project, pname) = 50,000
V(Employee, lname) = 250,000	V(Project, budget) = 50,000

Furthermore:

- Projects are uniformly distributed between 1861 (i.e., when UW was founded) to 2017.
- There is only 1 project for each employee (i.e., their "main" project), and each employee only worked on 1 project.
- Employees' birth years are evenly distributed between 1800 to 2000.
- 50% of Person has fname 'Dubs' while 0.0001% has fname 'Zzyzx'.
- Values for the other attributes are evenly distributed.

a) Consider the following query:

```
SELECT E.lname, E.fname
FROM Employee AS E
WHERE E.fname = 'Kodiak'
```

You can only create **one unclustered index** to speed up this query. Write down the attribute(s) and the type of index that you would create. (4 points)

Attribute(s): Employee.fname

Type: Hashtable

Information repeated here for your convenience.

```
Employee(SSN, yearBorn, fname, lname)
Project(PID, SSN, yearStarted, pname, budget) -- SSN is FK to Employee
```

And we have the following meta-information:

Employee	Project
T(Employee) = 200,000	T(Project) = 200,000
B(Employee) = 10,000	B(Project) = 10,000
V(Employee, fname) = 200	V(Project, pname) = 50,000
V(Employee, lname) = 250,000	V(Project, budget) = 50,000

Furthermore:
- Projects are uniformly distributed between 1861 (i.e., when UW was founded) to 2017.
- There is only 1 project for each employee (i.e., their "main" project), and each employee only worked on 1 project.
- Employees' birth years are evenly distributed between 1800 to 2000.
- 50% of Person has fname 'Dubs' while 0.0001% has fname 'Zzyzx'.
- Values for the other attributes are evenly distributed.

b) Consider the following query:

```
SELECT e.fname, p.pname
FROM Project as p JOIN Employee as e ON p.ssn = e.ssn
WHERE e.fname = 'Dubs' AND e.yearBorn > 1999 AND p.yearStarted > 1870
```

You can only create **one clustered** and **one unclustered** index to speed up this query. Write down the attribute(s) and the type of indexes that you would create. (8 points)

Clustered index attribute(s):
Employee.yearBorn, since selectivity of yearBorn > 1999 is 1/200.

Type: Hashtable / B+-tree

Unclustered index attribute(s): Employee.ssn or Project.ssn. After applying the selection predicates we expect 200,000/200 = 1000 tuples from employees. Creating index on Project wouldn't help since the query is pretty much selecting all projects. To speed up the query we should create an index on ssn from either relation.

Type: Hashtable

Information repeated here for your convenience.

```
Employee(SSN, yearBorn, fname, lname)
Project(PID, SSN, yearStarted, pname, budget) -- SSN is FK to Employee
```

And we have the following meta-information:

Employee	Project
T(Employee) = 200,000	T(Project) = 200,000
B(Employee) = 10,000	B(Project) = 10,000
V(Employee, fname) = 200	V(Project, pname) = 50,000
V(Employee, lname) = 250,000	V(Project, budget) = 50,000

Furthermore:

- Projects are uniformly distributed between 1861 (i.e., when UW was founded) to 2017.
- There is only 1 project for each employee (i.e., their "main" project), and each employee only worked on 1 project.
- Employees' birth years are evenly distributed between 1800 to 2000.
- 50% of Person has fname 'Dubs' while 0.0001% has fname 'Zzyzx'.
- Values for the other attributes are evenly distributed.

c) Now assume the workload consists of equal numbers of the following three queries:

```
SELECT p.pname
FROM Project as p JOIN Employee as e ON p.ssn = e.ssn
WHERE p.yearStarted > 2015 AND e.yearBorn > 1999
ORDER BY e.yearBorn

SELECT p.pname, p.yearStarted
FROM Project as p
WHERE p.name = 'Awesome Project'

SELECT p.pname, p.yearStarted
FROM Project as p JOIN Employee as e ON p.ssn = e.ssn
WHERE e.fname = 'Zzyzx'
```

Suppose you can create **one clustered index** for Q1, **one unclustered index** for Q2, and **one unclustered index** for Q3. Write down the attribute(s) and the type of indexes that you would create. (12 points)

Q1: Clustered index attribute(s): Project.yearStarted or Employoee.yearBorn, as the predicates on both attributes are highly selective.

Type: B+ tree

Q2: Unclustered index attribute(s): Project.name, and optionally on Project.yearStarted as well

Type: Hashtable

Q3: Unclustered index attribute(s): Employee.fname, since ‘Zzyzx’ is a rare name

Type: Hashtable

d) Suppose we have three relations, `A(a)`, `B(b,c)`, and `C(d)`, and the optimizer chose the following physical query plan to join the three relations together:

```
T_1 = A Join[a=b] B       // multi-pass hash join
Ans = T_1 Join[c=d] C     // nested loop join
```

Assuming that the optimizer has perfect knowledge about table statistics of `A`, `B`, `C`, the size of T_1, selectivities, and independence of the join predicates. Select either True or False for each of the following. You get 2 points for answering it correctly, -1 point for an incorrect answer, and 0 point for no answer. The minimum you will get for this entire problem is 0. (10 points total)

i) `C` does not fit in memory. True [✓ False]

ii) $T(A \bowtie B) \leq T(B \bowtie C)$. [✓ True] False

iii) Neither A nor B fits in memory. [✓ True] False

iv) `T(C)` is very large compared to $T(T_1)$. True [✓ False]

v) There are no index on any of the tables, since index-based algorithms are not used.

True [✓ False]

Problem 7: Distributed Data Processing (25 points total)

In computational linguistics, a **bigram** is a sequence of two words that occurs sequentially in the text (e.g., "we love databases" contains two bigrams, "we love" and "love databases"). In this problem, we will compute the **conditional probability** (CP) of the different bigrams, i.e., if you saw the word "we," what is the probability that "love" will immediately follow it? Mathematically this is defined as:

$$\text{CP}(w_i \text{ follows } w_{i-1} \text{ given that we just saw } w_{i-1}) = \frac{\text{Probability}(w_i \text{ follows } w_{i-1})}{\text{Probability}(w_{i-1})}$$

Given a corpus of documents, we can approximate the probabilities above using word counts:

$$\text{Probability}(w_i \text{ follows } w_{i-1}) = \frac{\text{count of bigram } (w_{i-1}, w_i) \text{ in corpus}}{\text{total number of bigrams in corpus}}$$

and

$$\text{Probability}(w_{i-1}) = \frac{\text{word count } w_{i-1} \text{ in corpus}}{\text{total number of words in corpus}}$$

So, if our corpus consists of two sentences: "we love databases" and "we love SQL," then

$$\text{CP}(\text{"databases" follows "love" given that we just saw "love"})$$

$$= \frac{\text{Probability}(\text{"databases" follows "love"})}{\text{Probability}(\text{"love"})}$$

$$= \frac{[\text{ count of ("love" "database") in corpus / total number of bigrams in corpus }]}{[\text{ word count of "love" in corpus / total number of words in corpus }]}$$

$$= \frac{[1/4]}{[2/6]} = 0.75$$

a) Compute the conditional probability CP function above using MapReduce. As in lecture, we provide the initial call to `map` below. You are free use multiple stages of `map` and `reduce` calls as needed, and you don't need to pair up each `map` with a `reduce` call. Don't worry about getting syntax to be exactly correct. (20 points)

```
// Many answers are possible. Here is a sample solution

void map(String key, String value) {
  // key : document name
  // value: document contents, use value.split(" ") to return an array of
  // words contained in the document
  words = value.split(" ")
  for (int i = 0; i < length(words); ++i) {
    emitIntermediate("total word count", 1);
    emitIntermediate(words[i], 1);
    if (i < length(words) - 1) {
      emitIntermediate(words[i] + " " + words[i+1], 1); // bigram count
      emitIntermediate("total bigram count", 1);
    }
  }
}

void reduce(String key, int [] counts) {
  int sum = 0;
  for (c : counts)
    sum += c;

  // map all counts to the same key, but remember the original key
  // associated with it
  emitIntermediate("dummy", [key, sum]);
}

void map(String key, String value) {
  // build a hashtable that stores the single word and bigram counts,
  // along with the total number of words ("total word count"), and
  // the total number of bigrams (i.e., "total bigram count")
  ht = [];
  vs = value.split(" ");

  for (int i = 1; i < length(vs); ++i) {
      ht.insert(vs[i-1], vs[i]);
  }

  int wordCount = ht["total word count"];
  int bigramCount = ht["total bigram count"];
  // now go through all bigram keys in the hashtable, and emit
  // the appropriate counts (can also iterate over value as well
  // rather than building a hashtable)
  for (k in ht) {
    ws = k.split(" ");
    if (length(ws) == 2) // it is a bigram
      // ht[k] is the count for bigram k,
      // ht[ws[0]] is the count for the first word in the bigram
      emitIntermediate(ws, [ht[k], bigramCount, ht[ws[0]], wordCount])
  }
}

void reduce(String key, int [] counts) {
  emit(key, (counts[0]/counts[1]) / (counts[2]/counts[3]));
}
```

b) Explain in one sentence how your program above achieves parallelism. (5 points)

We compute the word counts in parallel and also the final conditional probabilities in parallel using the two `reduce` calls.

Problem 8: Trivia! (2 points total)

Write something that'd make us laugh!

:)

-- END OF EXAM --
-- Thank you for taking this class. Hope you learned a lot. --
-- Enjoy your summer! --