Solution

INF 551 – Fall 2015

Final Exam (100 points)

Exam time: Dec. 13, 11:59pm – Dec. 14, 11:59pm (i.e., 24 hours)

You are to work on this exam on your own. You are not to share, email or discuss in any way the contents of the exam prior to Dec. 14, 2015. Failure to follow these rules will result in a failing grade for the *class*.

You can complete this exam by hand or electronically. If done by hand, please scan and generate a PDF. Result must be submitted via blackboard. Remember, if it is hard for me to read, it will be hard for me to grade!!

Exam must be submitted to blackboard by 11:59 PM (Midnight) on Dec. 14, 2015. We will deduct 10% from your total score for every 30 minutes that your exam is late.

Please fill in below:

I certify that I the work on this exam is only mine and agree not to discuss the contents or answers with my classmates until Dec. 14th. I understand that failure to follow these rules will result in me receiving a failing grade for the class.

Name, Date

1. [B+-tree, 15 points] Consider the B+-tree shown below. The tree has a degree of 2, that is, each node except for the root, should have at least two keys and at most four keys. Note that each node is labelled, e.g., the root is the node 'A'.

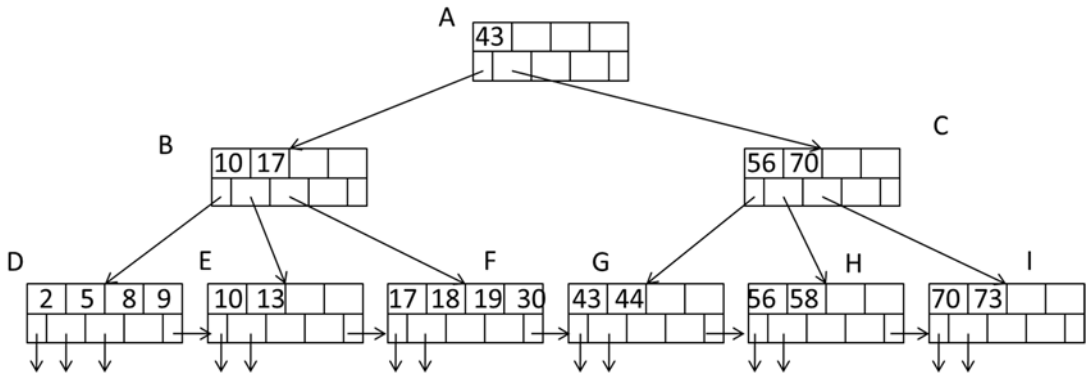


Figure 1: A B+-tree

   a. Which nodes are at their maximum capacity? Which nodes have just the minimum required number of keys?

Solution

Answer: maximum: D and F; minimum: B, C, E, G, H, I (could also say A, the root)

b. Explain why there is a requirement on the minimum number of keys in the node, e.g., the minimum number is 2 in this example. Explain why the root is allowed to be an exception.

Answer: this is to ensure that the tree is balanced. If there are nodes with a few keys and others with many keys, it will not be balanced. An unbalanced tree typically would have more levels than balanced one, thus increasing the cost of lookup.

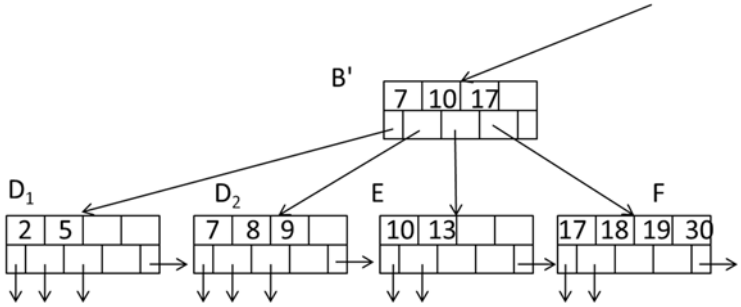
Root is exception since there can be only one key in the entire tree. (Accept other reasonable explanations too.)

c. When inserting a new key into a node X in the tree which is at its maximum capacity, there may be two options. One is to split the node X; the other is to move one of X's keys to its immediate siblings (i.e., nodes that are right to X and have the same parent as X) if they still have the room.

Now consider inserting 7 into the tree. For each of the two options, show the updated tree obtained using the option. Recall that when splitting X into X1 and X2, X2 (the right node) will have one more key than X1, the left. (It is ok to show the portion of the tree that has been changed due to the insertion.)

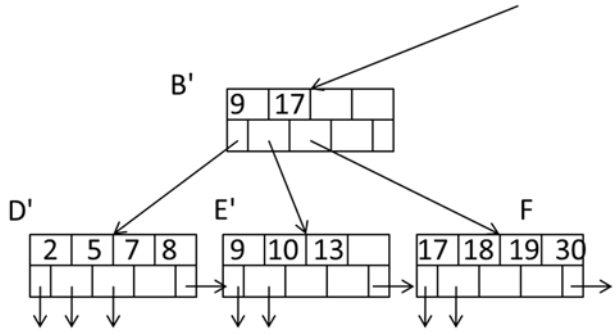
Answer:

(Splitting) The following is what we have with the splitting option. Note that D is split into D_1 and D_2. Furthermore, 7 was inserted into B which becomes B'.



(Moving) The following is the new tree after moving 9 to D's right sibling E which became E'. Note that 10 in B was replaced with 9.

Solution



d. Continue from the question c: Suppose only the root of the tree is in main memory and each node occupies a block on disk. For each of the options, how many block I/O's (read and write of a block each counted as one block I/O) are required to perform the insertion? Explain which blocks (i.e., nodes) need to be loaded into the memory and what nodes need to be written back to disk.

Answer:

For splitting option, nodes B and D need to be read into memory. D1, D2 and B' (new B) need to be written back to disk. So total of 5 block I/O's.

For moving option, B, D, and E need to be read into memory. B', D' and E' need to be written back. So 6 block I/O's.

e. At most how many **additional** keys may be inserted into the tree shown in Figure 1 (which has 3 levels), without increasing the number of levels of the tree? (You can assume that the keys are real numbers.)

Answer: Note that the keys are to be inserted into the leaf nodes of the tree. Keys in the internal nodes are there for the search purpose.

The maximum number of leaf nodes at a 3-level tree (with the degree of 2) is 5 * 5 (5 children of the root and each child has 5 children of its own, which are leaves). So the total of 25 * 4 = 100 keys.

Since there are already 16 keys in the leaves, at most **84** more keys can be inserted without increasing the level.

2. [Query Execution, 15 points] Consider joining two relations R and S using partitioned hash join. Assume that R has **200,000** blocks, S has **1000,000** blocks, and main memory has 101 pages (i.e., blocks). Loading of all tuples in R/S from disk is counted as one pass through R/S.

a. Is it possible to perform the join in one pass? Explain your answer.

Answer: No, it is not. One-pass will require either R or S to fit entirely in memory.

b. Is it possible to perform it in two passes? Explain your answer.

Answer: No, it is not either. Two-pass would require the memory to have at least sqrt(min(R, S)) ~ 448 blocks. Put it in another way, two-pass would require R or S to have at most 100^2 or 10,000 blocks.

Solution

c. Describe the step-by-step process of join (at each step, describe what is does, input and output, what are their sizes, etc.). How many passes does the algorithm make through R and S? What are the total block I/O's? (You may ignore the I/O's of writing join results on disk.) Do you have to use different hash functions in different passes? Explain your answer.

Answer: First, we hash R and S into 100 buckets each. We use one buffer (page) for the input and the remaining 100 buffers for the hash buckets. R's buckets will each have (about) 2,000 blocks, and S's buckets will each have 10,000 blocks. **Block I/O's: 2B(R) + 2B(S)** (for read R and S and write their buckets back on disk).

Next, we join the buckets of R with the corresponding buckets of S, that is, R1 with S1, R2 with S2, and so on. We may use a two-pass hash join now, since R's bucket (and S's bucket) is less than 10,000 blocks. Here are the details. For each Ri and the corresponding Si bucket:

- First, we hash Ri into 100 buckets, 20 blocks/each bucket. Similarly, we hash Si into 100 buckets, 100 blocks/each bucket. **Block I/O's (for all Ri and Si): 2B(R) + 2B(S)**
- Next, we join buckets Rij with Sij in one-pass. **Block I/O's: B(R) + B(S)**

Total block I/O's: 5B(R)+5B(S) = 5 * 1,200,000 = 6,000,000

Yes, different hash functions need to be used. Otherwise, all tuples in the subsequent hashing will be hashed into the same bucket.

3. [Data Warehousing, 20 points] Consider building a sales data warehouse using the data in Tables 1—3. Table 1 records the number of products sold to customers (e.g., three p1's sold to customer c1). Tables 2 and 3 provide details about products and customers.

ProdID	CustID	Qty
p1	c1	3
p1	c2	2
p2	c3	3
p3	c1	1
p3	c3	7

Table 1: Sales

ProdID	Category
p1	HDTV
p2	GPS
p3	HDTV

Table 2: Products

CustID	City	State	Region
c1	Charlotte	NC	East
c2	Raleigh	NC	East
c3	Los Angeles	CA	West

Table 3: Customers

a. Design a star schema for the warehouse. Indicate primary and foreign keys in each table of the schema.

Answer: The schema has sales as the fact table; products and customers as the dimension table. ProdID is the primary

Solution

key in Products; CustID in Customers; ProdID and CustID the (composite) primary key for the Sales table.
ProdID is a foreign key in Sales that refers to ProdID in Products. CustID is also a foreign key referring to CustID in Customers.

b. Are there meaningful hierarchies that exist among attributes in dimension tables? If yes, what are they?
Answer: yes. Products: Category -> ProdID (Category is on the top level of the hierarchy). Customer dimension: Region -> State -> City -> CustID.

c. Assume that functional dependency "state → region" holds for the Customers table. Is the table in BCNF? If yes, explain why. If not, decompose it into tables in BCNF.
Answer: No, it is not. Split it into two tables: (state, region), (CustID, city, state).

d. Explain the differences between star and snowflake schema and why star schemas are often used in practice despite the possible data redundancy it causes.
Answer: dimension tables in snowflake schema are in BCNF, and thus no redundancy in data. There is a single table per dimension in the star schema, which may store data redundantly, e.g., region is repeated for every occurrence of state in the region in this example.
However, redundancy tends to be less of an issue here due to several reasons.
(1) Dimension tables are small compared to fact table, so saving in storage is less significant.
(2) Dimension tables change less frequently than fact table, so update/insertion/deletion anomaly is not as big an issue.
(3) It is expensive to process queries using snowflake since it requires costly join operation to combine the multiple tables (which are merged into a single one in the star schema) for each dimension.

e. Consider building a bitmap index for **category** of products. State the bit-vectors in the index.
Answer: HDTV: 101; GPS: 010

f. Consider building a bitmap **join** index for **category** of products. State the bit-vectors in the index.
Answer: HDTV: 11011; GPS: 00100

g. Explain bitmap index is suitable for what kind of attributes. Is it a good idea to build a bitmap index for ProdID attribute of the Products table? Explain why.
Answer: attributes with small number of values in their domains (i.e., low cardinality).
No, the attribute is key, thus # of bit vectors is the same as the number of row. If there are n records, the space of bitmap will be n^2, among them only n bits are 1's. Thus, a very sparse bitmap and not space-efficient.

Solution

4. [NoSQL & Big Data, 25 points]
   a. [Cassandra] Explain the "consistent hashing" idea in Cassandra, what advantages it has, and how it addresses the problem with the traditional hashing.

      Answer:

      Consistent hashing is used to determine which keys need to be assigned to which nodes in the cluster. In traditional hashing, almost all keys will need to be assigned to a different machine when a new machine is added to the cluster. For example, going from n to n + 1 machine, the machine number will change from h(x) % n to h(x) % (n+1). Or you may think of hash value of a key x as a number between 1 to the number of machines (which changes from n to n+1 here).

      In consistent hashing, keys are hashed into a circle of length 1 instead (i.e., hash value is between 0 to 1). In other words, hash value does not change when new machines are added.

      Machine numbers are hashed into the same circle too. As such, their hash values do not change either when new machines are added.

      Keys are then assigned to the machine which is located right after them in the clockwise position.

      When new machine is added, only keys that fall into the range between the position of new machine and the one before it need to be reassigned to the new machine.

      This avoids the large amount of data movement in the traditional hashing method.

   b. [HBase] Describe the steps involved in writing data in HBase, e.g., executing a 'put' command. Make sure you indicate the order of the steps.

      Answer:

      First, edits are written into WAL log to ensure recovery.

      Next, new data are written into MemStore (in memory).

      (Students may also mention that WAL logs need to be replicated in multiple machines.)

      Finally, client is acknowledged of the completion of write.

   c. [BigTable] What are the concepts/components in HBase that are similar to "tablets", "tablet servers", "memtable", "SSTable Files" in the Google's BigTable (refer to Sec. 5.3 of the paper "BigTable: A Distributed Storage System for Structured Data")? Explain why it is necessary to perform minor and major compactions according to the BigTable paper. What are the major differences between the two?

      Answer: tablets are called regions in HBase; tablet server region servers; memTable called memstore; SSTable files are HFiles in HBase.

      Minor compaction writes the content of (full or over threshold) memTable to disk as SSTable. It reduces the main memory usage and the amount of recover work (since once

Solution

data are written on disk, it will not require replay/redo of WAL logs for them.)
Major compaction merges all SSTables (typically for a column family) into a single SSTable periodically. This avoids the growth of # of SSTables out of bound (due to frequent minor compactions). It also removes deleted data (which are still there since only they were only marked as deleted). It thus reclaims the storage occupied by deleted data and makes sure deleted data disappear from the system in a timely fashion.

d. [Pig Latin] Which transformation command in Pig Latin is similar to selection in relational algebra? What about projection?
Answer: filter command is similar to selection. Foreach … generate (or simply foreach) can perform projection.

e. [Hive] Briefly describe the steps in processing a query in HiveQL in Hive.
Answer: First, query is complied into a logical data plan; next, logical plan is optimized by a rule-based optimizer into a DAG of mapreduce and hdfs tasks; finally, tasks are executed by an execution engine.

5. [Apache Spark, 25 points] Recall from your homework 5 that HITS algorithm computes authority and hub scores of nodes in a graph using mutual recursion. In this case, we consider only one iteration of the algorithm that starts with initial authority scores (all 1's) and computes the hub score of each node. Recall that the hub score of a node X is the sum of the authority score of the nodes X points to.

   You are provided with the following Python script running on Apache Spark, which computes the hub scores of the following directed graph (Figure 2).

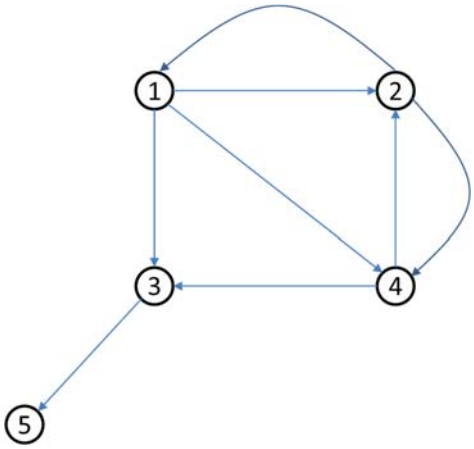


Figure 2: A directed graph

Solution

```
def parseEdges(edge):
   nodes = edge.split(',')
   return (int(nodes[1]), int(nodes[0]))

def contributeScoresList(nodes, auth):
   res = []
   for node in nodes:
         res.append((node, auth))
   return res

lines = sc.textFile('graph.txt')

edges = lines.map(lambda l: parseEdges(l)).cache()

links = edges.groupByKey().cache()

auths = links.map(lambda neighbors: (neighbors[0], 1.0))

joined = links.join(auths)

hubsMap = joined.flatMap(lambda t: contributeScoresList(t[1][0], t[1][1]))

hubs = hubsMap.reduceByKey(lambda x, y: x + y)
```

The script reads a text file “graph.txt” where each line corresponds to an edge in the graph of Figure 2. For example, the first line “1,2” says that there is a directed edge going from node 1 to node 2 in the graph.

graph.txt:

```
1,2
1,3
1,4
2,1
2,4
3,5
4,2
4,3
```

Answer the following questions on the execution of the script.

a. What is an RDD in Spark? What do we say RDDs are distributed & resilient? Name at least one RDD in the above script.

Solution

Answer: RDD is a Resilient & Distributed Dataset. It is distributed since Spark divides the dataset into several partitions and distributed over the nodes of a cluster to be processed in parallel. It is resilient since Spark keeps track of transformations on the dataset which enables efficient recovery on failure.
All variables in the main body of the script (lines to hubs) are RDDs.

b. What is the value of the variable "edges"? What does the "map" method in lines.map(…) do in Spark?
Answer: [(2, 1), (3, 1), (4, 1), (1, 2), (4, 2), (5, 3), (2, 4), (3, 4)] (note the tuples in edges may show up in different order as shown here)
"map" method here transforms each element in the lines RDD by parsing it from a string, e.g., "2,1" to a tuple, e.g., "(2, 1)".

c. What does the "cache()" method do in Spark?
Answer: It caches the content of RDDs in main memory for reuse.

d. What is the value of "links"? What does the "groupByKey" method in edges.groupByKey() do?
Answer: [(1, [2]), (2, [1, 4]), (3, [1, 4]), (4, [1, 2]), (5, [3])]
It groups the key-value pairs in edges RDD by keys. For each group/key, it produces a list of values with the key. Note that since parseEdges function returns tuple in reverse order, it gives reverse edges, i.e., it will return (2, 1) for an edge (1, 2) as given in graph.txt. Note that this is different from parseEdges function in the homework. So links basically list for each node X, a list of nodes that point to X (and thus contribute to X's authority).

e. What is the value of "joined"? (Here you need to provide the actual join result. It is NOT ok to simply say it contains a ResultIterable object as Spark will report if you do a simple collect() action on joined. Instead, state what is the content in the ResultIterable object.) What does the "join" method in links.join() do?
Answer: [(2, ([1, 4], 1.0)), (4, ([1, 2], 1.0)), (1, ([2], 1.0)), (3, ([1, 4], 1.0)), (5, ([3], 1.0))]
It joins the links RDD with auths RDD (both are key-value pairs) by their keys.

f. What is the value of "hubsMap"? What does the "flatMap" method in joined.flatMap() do?
Answer: [(1, 1.0), (4, 1.0), (1, 1.0), (2, 1.0), (2, 1.0), (1, 1.0), (4, 1.0), (3, 1.0)]
flatMap maps each joins tuple into a list (students may simply copy from Spark documentation to say that 0 or more output items. Accept it too.) Note that this is done via contributeScoreList function which takes the second

Solution

component (which is also a tuple: a list of nodes, and a hub score) of the joined tuple. The function returns a list and flatMap combines a list of lists into a single list.

g. What is the value of "hubs"? What does the "reduceByKey" method in hubsMap.reduceByKey() do?

Answer: [(2, 2.0), (4, 2.0), (1, 3.0), (3, 1.0)]
reduceByKey takes a list of key-value pairs, group the pairs by keys, apply a function (addition in this case) to add up all values with the same key.