Principles of Computer System Design (PCSD), Block 2, 2012/2013 Final Exam

Tzemis Evangelos
22/01/2013

Questions

Question1: In this part I will try to implement a single-threaded Map Reduce runtime that operates on large data in external memory. For this reason we have to make the assumption that we have an input File that contains all the data we want to Map Reduce, and the file is accessible by our system. In order to answer in detail, I will first explain my implementation and then present my pseudo code.

As it should be Single-Threaded we will have only one worker, which will first operate as a Map Worker and then as a Reduce Worker. Furthermore we want the runtime to be able to operate on large, disk-resident data collections that exceed the main memory's size. For that reason we will need to split the input file to smaller ones that will easily fit into the main memory. After splitting the file we will call the map function for every part of it. Each time the map function will output some Intermediate Pairs (k,v) in another file in the system. Every time it will be called again it will append the pairs to this file.
After the mapping is finished we will continue to the next step, which is to sort all the data in order to have all values for each key together. The tough thing is that the values are in the disk and not in the memory. So we are going to do the external sorting and then the grouping. After the external sorting the sorted file will be like this:

(k1,v1)(k1,v2)…(k1,vn)(k2,v1)..(k2, vn)……(km, vn)

With grouping we are going to merge all values for a specific key together. So we will have this type of file:

(k1, v1,v2,..vn)(k2,v1,v2,..vn)….(km,v1,v2,..vn).

After that we are ready for the Reduce Phase. We are going to feed reduce function one pair per time. In this way it will be able to fit in memory. This time the reducer will output (appending at the same time) the results into the Output File. After the reducer finishes reading all the (Key, List (Values)) pairs it terminates.

```
MapReducer Pseudocode

fileName = ReadinputFileName;
NoPiecies = GetSize(fileName)/64Mb;
SplitIntoPiecies(NoPiecies);
//Create New Intermediate File
InterFile = new File();
For each piece do{
     f = put file i in memory;
     //run Map function with this file as input;
     intermediatevalues = map(f);
     //append intermediatevalues to IntermediateFile;
     InterFile.append(intermediatevalues);
}
//Now we have all intermediate data in this file
//External sorting based on key
ExternalSorting(InterFile);
//Group values with same key into the same record
Group_by_key(InterFile);
//Create New Output File
OutFile = new File();
//feed each Record to the Reduce Function
For each record in InterFile do{
     //run Reduce function with this record as input;
     finalvalues = reduce(record);
     //append finalvalues to OutFile;
     OutFile.append(finalValues);
}
exit;
```

c) As mentioned above after map phase finishes we will sort all pairs by key. So no matter if map face skews the keys, we will put them in the correct order for the reduce phase. However, if they are sorted then sorting will be much faster that in the case of unsorted. So yes, we tolerate this situation, however performance is affected.

Question 2 (Serializability & Locking): Consider the following two transaction schedules with time increasing from left to right. C indicates commit.

Schedule 1:

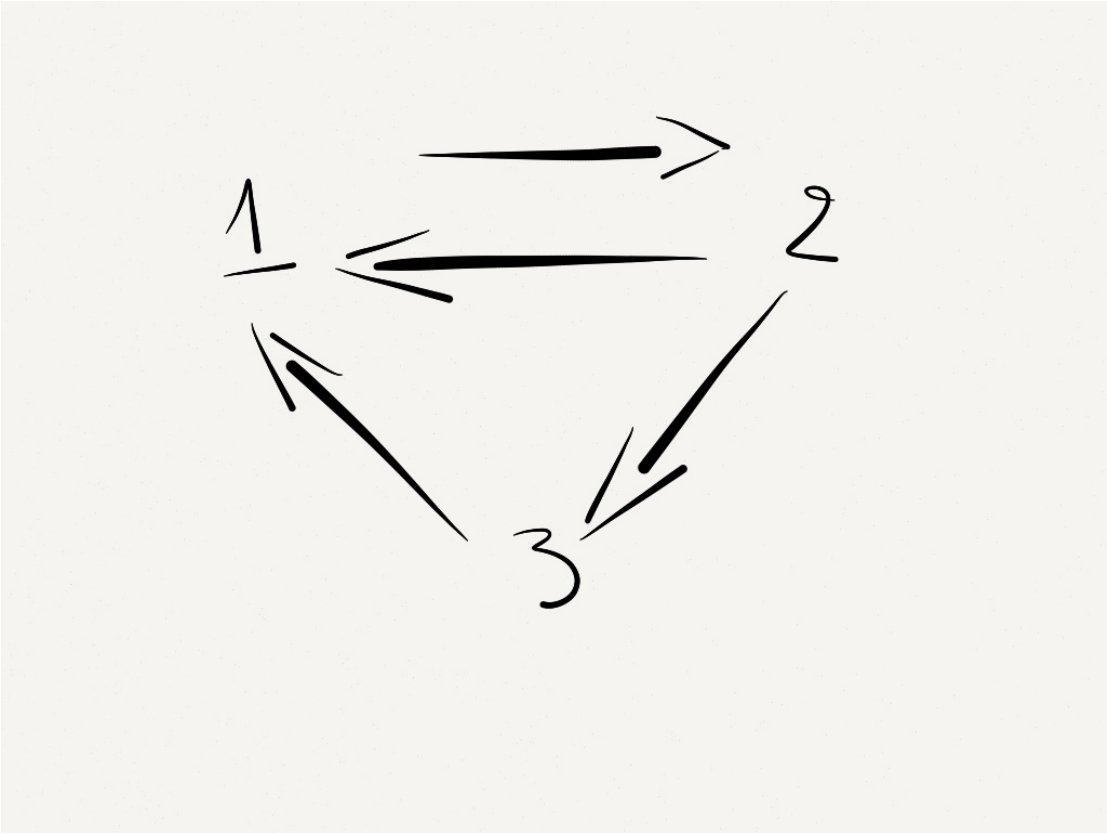
T1: R(X)			W(Y) C	
T2:	W(X) W(Y)			C
T3:		W(Y)		C

Schedule 2:

T1: R(X)			W(Y) C
T2:	W(Y) W(Z) C		
T3:		W(Y) C	

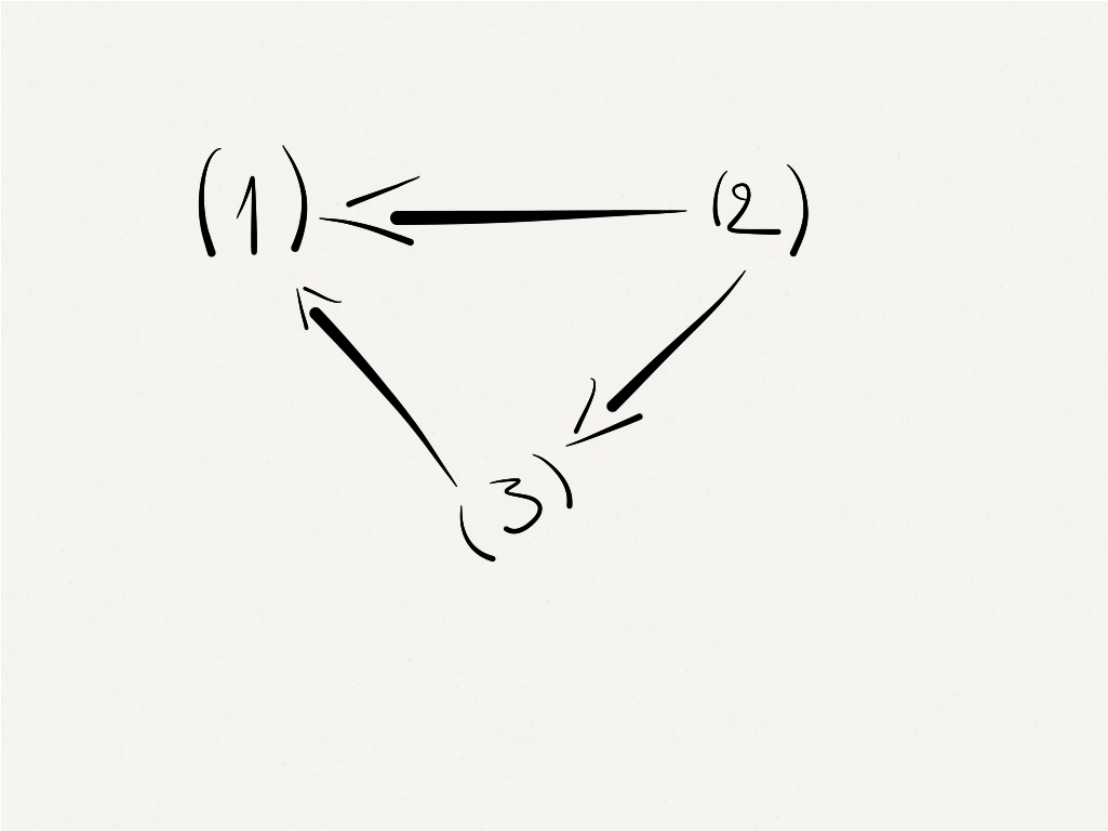
(a) Draw the precedence graph for each schedule. Are the schedules conflict-serializable? Explain why or why not.

Precedence Graph for schedule one is:



As we can clearly see this schedule is not conflict serializable because as the theorem states a schedule is conflict serializable if and only if its dependency graph is acyclic. However we can see that in the above graph there is a circle (2-3-1-2) or (2-1-2).

Precedence Graph for schedule two is:



In this occasion and for the above-mentioned reason this schedule is conflict serializable because its dependency graph is acyclic.

(b) Could the schedules have been generated by a scheduler using strict two-phase locking (strict 2PL)? If so, show it by injecting read/write lock operations in accordance to strict 2PL rules. If not, explain why.

Schedule 1:

As we know, in Strict Two Phase Locking every operation must obtain a shared lock (exclusive) on an object before reading (writing). All locks held by a transaction are released when the transaction commits. Also a strict 2PL allows only schedules whose precedence graph is acyclic, so

for this particular occasion, a S2PL scheduler could not have generated the schedule, because the precedence graph is cyclic.

Schedule 2:

Question 3 (Recovery):

A.

(1) The state of the transaction and dirty page tables after the analysis phase

Transaction table		
Fist Occurrence	Last LSN	Status
(T1, LSN 3)	3	U (to be undone)
(T2, LSN 4)	11	U

As stated in the compendium, we scan the log in the forward direction until we reach the end of the log. T3 is not present in the above table because we encountered an end log record for that transaction.

Dirty Page Table	
Page	LSN
P1	3
P1	4
P3	6
P3	7
P5	8

(2) The sets of winner and loser transactions

Winner = {T3} and Loser = {T1, T2}

(3) The values for the LSNs where the redo phase starts and where the undo phase ends.

Aries can determine where REDO phase starts by finding the smallest LSN, M, of all the dirty pages in the Dirty Page Table. So the REDO phase starts at LSN 3.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions has been previously identified in the Transaction

Table during the analysis phase. Now, the UNDO phase proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo set has been undone. So Undo phase ends at LSN 3.

(4) The set of log records that may cause pages to be rewritten during the redo phase.

During the Redo phase, ARIES reapplies the updates of all transactions, committed or otherwise. So in this case ARIES is going to reapply all the updates contained in dirty page table.

Page	LSN
P1	3
P1	4
P3	6
P3	7
P5	8

(5) The set of records undone during the undo phase.

The set that will be undone are those belonging to loser transactions that modify pages. So it is

Set of log records UNDO	
Page	LSN
P1	3
P1	4
P7	5
P3	7

(6) The contents of the log after the recovery procedure completes

The last question is basically the log as it was presented, together with the clear records from loser transactions. Those records contain the UndoNext field and of course the page they are modifying. So it is the same as in the beginning but all processes except for the T3 (which ends) will have the Undonext field equal to true.

LSN	LAST_LSN	TRAN_ID	TYPE	PAGE_ID	UNDONEXTLSN
1	-	-	Begin CKPT	-	-
2	-	-	End CKPT	-	-
3	NULL	T1	Update	P1	NULL
4	NULL	T2	Update	P1	NULL
5	4	T2	Update	P7	4
6	NULL	T3	Update	P3	
7	5	T2	Update	P3	
8	6	T3	Update	P5	
9	8	T3	Commit	-	
10	7	T2	Abort	-	
11	10	T2	CLR	P3	5
12	9	T3	End	-	

B. In the example above, you have shown the values for the LSNs where the redo phase starts and where the undo phase ends. State the conditions that characterize these LSNs in general, independent of the scenario above. In other words, how far back into the log must ARIES scan during the redo and undo phases of recovery?

As far as REDO phase is concerned we have the following. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to the dirty pages have already been applied to the database on disk. It can determine this by finding the smallest LSN, M, of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to a LSN < M, for redo-able transactions, must have already been propagated to the disk or already been overwritten in the buffer; otherwise, the dirty pages with that LSN would be in the buffer (and the Dirty Page Table).

In UNDO phase ARIEs works in the following way. An important thing to mention is that once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. Now, the UNDO phase proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo-set or the loser transactions has been undone.

https://www.classle.net/book/algorithm-aries

Implementation Part

High-Level Design Decisions and Modularity

Question 1: Describe how you organized data distribution in your implementation of the Account Manager and your implementation of enforced modularity. In particular mention the following aspects in your answer:

- **How you chose to partition branch data across nodes (e.g., round robin, hash, range) and justify why.**

As stated "You may assume in your implementation that the data is loaded once when each partition is initialized at its constructor". According to this I have a file called PartInput.txt, which I have organized in the following way:

Branch
AcountId *AccountAmount*
.
.
Branch
AccountId *AccountAmount*
.
.

According to this I am able to initialize my partition with as many Branches and Accounts as I want to. For load balancing issues I preferred to have the same balance in each Branch and the same amount of Branches in each Partition (25 accounts per branch and two branches per partition).

- **Which RPC mechanism you have chosen, along with advantages and disadvantages of this mechanism.**

I chose to implement this assignment using JAX-WS in order to expose my application as a service. The most significant reason for this choice was the experience we gained from the first three assignments. Apart from that, annotation support is important and it gives us the opportunity to annotate Java classes with metadata in order to indicate that the Java class is a web service. However, it is not always too trivial to implement a web service because when you want to pass classes through a function, things will become complicated because you have to convert from a source to a destination class.

Atomicity and Fault-Tolerance

Question 2: An AccountManagerPartition instance executes operations originated at multiple AccountManagerProxy instances. Describe how you ensured atomicity of these operations. In particular, mention the following aspects in your answer:

- **Which method did you use for ensuring serializability at each partition (e.g., locking, optimistic approach, or simple queueing of operations)? Describe your method at a high level.**

Although that there are many different ways for ensuring serializability, at each partition I chose to use locking. In this way every account object except for the ID and the amount contains a ReadWriteLock, which we used to ensure locking. So many requests may arrive at a specific branch for the same account. Instead of putting them in a queue, we let them run concurrently. However if they try to modify data they have to use a write lock, and if they try to read data they will have to use a read lock.

- **Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock.**

As we can easily understand the only thing that makes us use serializability methods is that we want to be able to modify account information. Trivial solution would be to use a global lock and make all operations atomic. However, this is a waste of time since we lock for a long amount of time. When the only locking needed is exactly when we change the data. In this way all locking before and after modifying is useless. And this is what I am doing minimizing critical section and also make atomic only the write functions. So equivalence between global and my locking is obvious.

- **Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other alternatives you considered (e.g.., locking, optimistic approach, or simple queuing of operations) would be better or worse than your choice. If possible, employ a back of the envelope calculation, or supporting numbers from your experiments.**

Hypothesizing that we are going to have enough requests for the same account, for example in the case of a full load we might have for instance ten parallel requests per account. These requests may vary between transfer (Origin or Destination one) Credits and Debits. Alternatives that I considered were simple queuing but I think that ReadWrtiteLock is fast enough. If we can assume that each of the transfer debit or credit takes a specific amount of time if it was executed atomically (for example c seconds), then in the case of 10 concurrent requests we have the following.
With simple queuing we would need approximately 10*c seconds as they were going to be executed one after the other. However, with my implementation only c + a constant value is going to be needed because they are going to execute almost in parallel (plus the time needed for the locking). After all I strongly believe that my implementation in heavy load is going to be significantly better. However, when there is only one request for a specific account simple queuing might have been a bit better because it does lock anything.

Question 3 (LG4): Describe how you have tested your service. In particular, mention the following aspects in your answer:

- **How you tested that operations were indeed atomic.**

Testing has been done with debugging. By that I mean that firstly I opened two clients and called a specific operation (e.g. Debit) for a specific account in a specific branch (the same in both of them). I had already run in debugging mode the Partition contained this specific branch and account. I placed breakpoint inside the account Class, at the points the value is changing. Then I let both request hit the breakpoint. Letting one go inside the critical section should keep the other outside, which actually happened. I did some other similar tests to ensure that atomicity is guaranteed under all circumstances.

- **How you tested that the system indeed respects a fail-soft design.**

Testing has been done again with debugging. I initialized the system and ran the Proxy in Debug mode. At some point I killed a partition. Till now everything is fine. After that I made a request from the client, which was supposed to address the killed partition. Then I checked whether the program worked correctly. And indeed it handled the exception correctly and changed the partition status to down, so no more requests will be addressed to this partition.

Experiments

Question 4 (LG6): Design, describe the setup for, and execute an experiment that shows how the throughput of the service scales with the addition of more partitions. Remember to thoroughly document your setup. In particular, mention the following aspects in your answer:

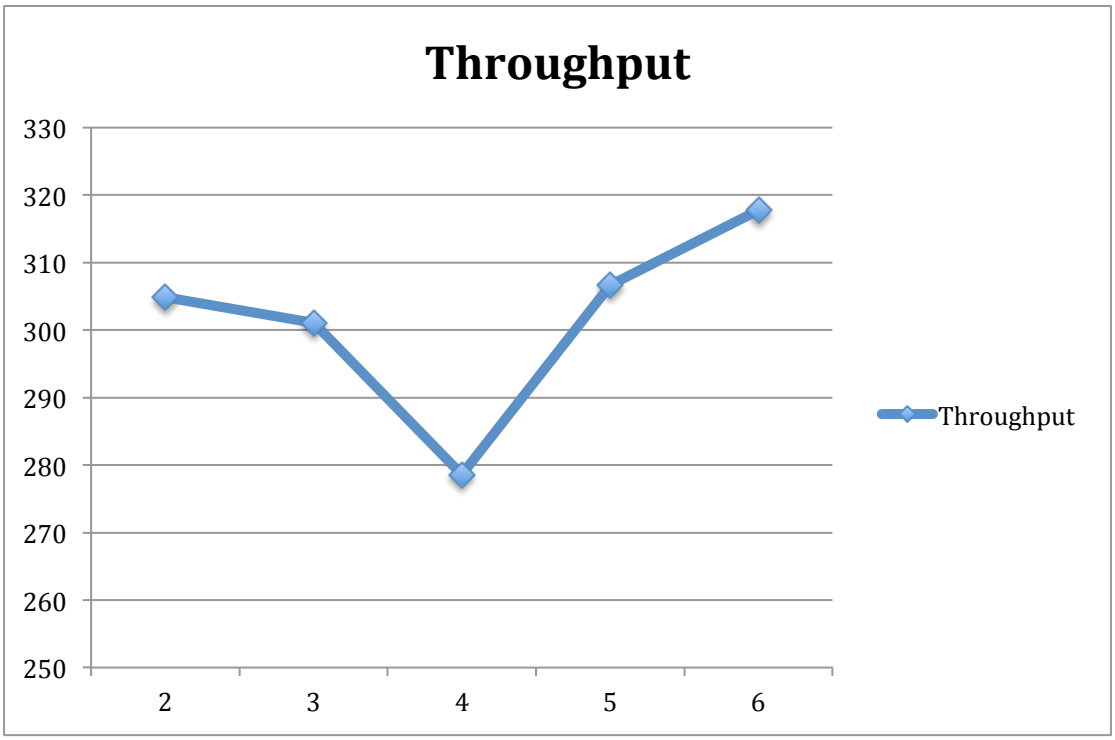
- **Setup: Document the hardware, data size and distribution, and workload characteristics you have used to make your measurements. In addition, describe your measurement procedure, e.g., procedure to generate workload calls, use of timers, and numbers of repetitions.**

I made test in my personal computer and I did not used the Microsoft Azure, so these are the hardware limitations. I made test with up to 6 partitions (from 2 to 6). Every partition for this test contained only 2 branches with 25 accounts per branch. So we had equal distribution of data to the partitions, which helped to provide more general testing results. In order to make the throughput measurement I made the Clients perform only credit request to a randomly choses branch and account. The proxy used a counter to count all credit requests. A thread was printing the average Throughput every 10 seconds. In each of the individual tests I used 8 clients (they ran infinite number of times (only credit)). For every partition I let the system run for 10 minutes with those 8 clients, to get reliable results.

- **Results: According to your setup, show a graph with your throughput measurements on the y-axis and the number of partitions on the x-axis. Describe briefly your results. How does the observed throughput scale with the number of partitions? Explain why**.

According to the tests I deed the throughput results are presented in the following table and graph.

Partition	Throughput
2	304.869697
3	301.055882
4	278.575676
5	306.707692
6	317.77541



In general I did not noticed any big difference when I was adding partition. Particularly, in the beginning we have a decrease to the throughput from partitions 2 to 4. This might have happened because of the higher communication cost. However adding more partitions made throughput increase again so I concluded that it might was mine hardware's issue. I run all test twice but with almost the same results. However I do not thing that adding partitions would actually influence throughput, case the only big deal with more partitions is that u need more time to search correspondence between branch and partition. However if the partitions where placed in big geographical distance then throughput would definitely be influenced by this distance.