ACS EXAM 2016

qlg389

June 26, 2019

1 Data Processing

1.1

Overview of algorithm

For the triple-click.com, I select a sort based algorithm to answer the k-closest event query. Let P and B denote the number of input pages to be read and the number of pages that can be read into main memory(i.e. we have B pages in main memory) respectively. This algorithm will employ a slight change in the use of buffers, the size of the input buffers will be reduced to allow for two output buffers. Although, this will not reduce the cost of I/O directly, it will speed up operations.

Phase 1 of the algorithm, is to read B pages of triple-click's scripts for each run and sort using replacement selection (a variant of the heap sort algorithm with a min-heap property) and then write out the sorted runs to disk. Replacement selection helps to reduce the amount of I/O operations by maintaining an input buffer and an output buffer, we read from disk only when the input buffer is empty and write to disk when the output buffer is full. Therefore, the number of runs is P/B, and the cost of phase 1 is 2P, which is the cost of reading input pages in and out of disk. We do not account for the cost of the sort because its of no consequence compared to the cost of I/O which is the main focus of external sorting algorithms.

Phase 2 of the algorithm, involves merging the runs to get the k-closest clicks for each event. We have B pages in main memory; we assign B-2 pages to merging and assign 1 page to one output buffer that holds a k-dimensional tree and assign the last page to another output buffer that stores the lists for the k-closest items.This is to reduce the number of times these lists are written to disk, however, it is possible that they are being processed directly by another application. In this case, this would not matter. Since all runs have been sorted, we merge them by building a k-dimensional tree (in an output buffer) using the min heap property where k is the number of the closest clicks we need to find. Once we have k+1 nodes in the tree, we start searching for the k-closest clicks of events starting from the root node.We use the best bin first algorithm when searching the tree for the k-closest clicks, this means we traverse the tree closest to the query point(the root node) until some computational budget is exhausted, then we report the k-closest click for the root node and copy the result into the second output buffer. When the second buffer is full, we copy all k-closest clicks to disk. The k-closest clicks are guaranteed to be children of the root node. We do this for all nodes and output the result in a list.

Algorithms

Algorithm 1: Pseudocode for replacement selection

```
1  Fill the array from disk. Set LAST = M-1.
2  Build a min-heap. (Recall that a min-heap is defined such that the
   record at each node has a key value less than the key values of its
   children.)
3  for all elements in array: do
4      Send the record with the minimum key value (the root) to the
       output buffer.
5      Let RR be the next record in the input buffer.
6      if RR 's key value is greater than the key value just output ... then
7          place RR at the root.;
8      else
9          replace the root with the record in array position LAST, and
           place RR at position LAST. Set LAST = LAST - 1.;
10     end
11     Shift down the root to reorder the heap.
12 end
```

Algorithm 2: Pseudocode for K-closest clicks

```
Data: triple-clicks.com scripts
Result: K-closest clicks for all events
for each chunk of B pages of N do
    sort using replacement selection;
    write to disk;
end
for all P/B sorted runs do
    if output buffer for storing tree is full then
        select k closest clicks for root node using best bin first
        algorithm;
        write tuple containing root node + k-closest clicks of root node
        to second output buffer;
        delete root node;
    else
        continue;
    end
    if output buffer for storing lists is full then
        empty buffer by writing to disk;
    else
        continue;
    end
    if there are k+1 nodes left in the tree and the sorted lists are empty
    then
        they are all k-closest of each other(write tuple for all k events);
        stop;
    else
        continue;
    end
    select the least event from the runs and place them in input buffer ;
    Add event to k-dimensional tree in the output buffer;
end
```

Correctness and Efficiency

The correctness of the first phase of the algorithm follows from the fact that replacement selection algorithm selects the minimum element and writes it

to the output buffer, therefore, we always end up with a sorted list.For the second phase, we select the minimum element from the runs and build a k-dimensional tree, the k-closest clicks of a node are guaranteed to be either the children or very close to the children.Although, the complexity of the algorithms involved is of no consequence compared to the cost of I/O.
The following are the costs of each of them
Sorting by replacement selection- O(n log n)
Building a k-dimensional tree- O(n log n)
Bin first search- O($n^{1-1/k} + m$) where k is the dimension of the tree and m is the number of elements you need to search for. Assuming the KD tree can fit in memory then this algorithm works optimally, we would not need to delete nodes; hereby using less time. The efficiency of this algorithm largely depends on P.

1.2

IO cost

Cost of phase 1 = 2P. Let N denote the number of passes to sort P pages. Then P = $B(B-2)^N$ and N = $[log_{B-2}[P/B]]$. This is because we sort B pages at a time in the first phase and merge them using B-2 pages. The cost of each pass = 2P, we write P pages in and out of disk when merging.
The cost of phase two is = $2P \times [log_{B-2}[P/B]]$
The total cost of phase 1 and 2 = $(2P([log_{B-2}[P/B]]) + 1)$

2 Concurrency Control

2.1

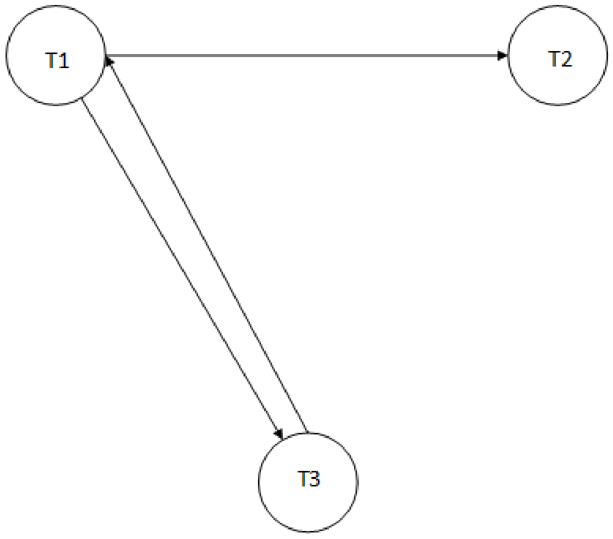
A schedule is view serialializable if it is view equivalent to some serial schedule while a schedule is conflict serializable if its precedence graph is acyclic. The schedule in Table 1 is view equivalent to $r_1(A)r_3(A)w_1(A)w_2(A)$ but not conflict serializable because it contains a blind write(this is when a transaction's last action was a write and the next transaction performs a write on the same object without reading it first, for example T1 and T2) and the precedence graph is not acyclic.

Table 1: View Serializable schedule

T1	T2	T3
R(A)		
		W(A)
W(A)		
	W(A)	

Figure 1 shows that is not conflict serializable since the graph is not acyclic; T3 has to wait for T1 to release the shared lock on A, T1 has to to wait for T3 to release the exclusive lock on A and finally T2 has to wait for T1 to release the exclusive lock on A.

Figure 1: Serializable precedence graph



2.2

Table 2 is conflict serializable as it is conflict equivalent to serial schedule: $w1(A)r1(A)r2(A)$. However, it is not cannot be generated by a two phase locking scheduler because in 2pl a transaction cannot request additional locks once it releases them(i.e. locks are not acquired in the shrinking phase).This

Table 2: Conflict Serializable

T1	W(A)		R(A)
T2		R(A)	

Table 3: Two phase locking

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	C
C	

means that once T1 releases the write lock on A it cannot acquire a lock again on A because it is in the shrinking phase.

2.3

In strict two phase locking, we release locks only after a transaction is completed, whereas in two phase locking we release before we are ready to commit.Table 3 cannot be generated by a strict 2pl because a transaction does not release locks until it is ready to commit.T2 cannot a lock on A until T1 has reached its commit phase.

2.4

In a conservative strict scheduler, we will need to predeclare all needed locks unlike in the strict scheduler where we lock objects as they are required. Table 4 shows an example of a strict two phase schedule that cannot be generated by a conservative strict protocol.This is because T2 should have acquired all its locks before starting; an exclusive lock on A and B.

Table 4: Strict two phase locking

T1	T2
R(A)	
	R(A)
	R(B)
	W(B)
	C
R(C)	
W(C)	
C	

3 Programming Task

3.1

Organization of modules and data

The code is organized into seven packages; business, client, interfaces,server, tests, utils and workloads.
The business package contains the implementation of the interface (Auction market) and the final classes for Item and Bid.
The client package contains the implementation of the auction market that allows the client to communicate with the server.SendAndRecv is used by the client to send and receive information from the server, POST method is used in sending information because it is more secure for sending sensitive information. For instance we do not want the bids of clients to be available to other clients.
The interfaces package contains the interface of the auction market.
The server package is where the server is started(certainMarketHttpServer), it responds to client requests using the message handler which handles all functions of the auction market.
The tests package contains tests that can be run either locally or using the server(by setting the variable localtest to true or false).
The utils package contains supporting code for the entire auction market.
The workloads package is where the experiment on the auction market is carried out(to measure the throughput of the system).

RPC semantics

The architecture implements the at-most-once semantic- either it does not execute at all or it executes exactly once depending on whether the server goes down. When the client sends a request it uses the *SendAndRecv* function which sends it and then waits until the response is available (waitForDone). Something could go wrong and in this case it just throws an exception (instead of sending it again and making sure it is done exactly once) so the request is done at most once.

All-or-nothing

I will discuss how all-or-nothing was implemented in each function of the Auction Market.

Add-items

Before items are added into the auction market, the items are validated;the program checks if the input is null, whether the item ID's are invalid, that the item description is null or empty and the item is in store. If any of these conditions returns true, then it throws an exception and no item is added. This ensures atomicity in that either all books are added(no exception) or no books added(in the case of an exception).

Bid

Before bids are made, the inputs are validated;check that the items to be bid on are in the market, a valid price is used and that the buyer organization Id is valid. Only after performing these checks are the bids added. To ensure atomicity of operations, if any of the input is invalid, none of the bids are made.

Switch Epoch

This is a rather sensitive portion of the system, if something goes wrong while switching an epoch , we could loose everything in a current epoch.However, having no data at all is better than having corrupted or incomplete data(purely a design decision). Therefore, I have decided that if something goes wrong while switching an epoch, the epoch should be cleared, in order to maintain atomicity. In any of these methods, if there is a failure, the system will be at its previous state i.e. none of the operations will be carried out.

Durability

The final matching state of the epoch is the most important one as long as it is successfully written to disk.But what if 500 organisations bid on an item, we declare a winner, we are writing to disk and something goes wrong; the program crashes. We loose everything, including the item that was being bid on! A better way of ensuring durability is to have a buffer where we only save items that are being bid on, when this buffer is full, we write them to disk. Also, before we switch an epoch, we empty this buffer. This ensures that if something goes wrong while switching the epoch, we can at least restart the epoch with "important items".This does not exclude the current method but combines the two for maintaining durability.

3.2

Serializability

I have made use of locking to ensure serializability of operations in the auction market. I have made use of a read lock for query items and write locks for switch-epoch, add-items and bid. My implementation ensures that a read and write lock cannot be obtained at the same time and only one write lock can be obtained at a time. This ensures serializability.

Correctness

The protocol used here is the conservative strict two-phase locking. It is conservative because the lock is acquired at the beginning of the method, just like in a conservative two-phase locking even though it is a single lock. It is strict because the lock is only released at the end of the method in the finally block, and not during execution. We release the lock only in the finally block because if an exception is thrown we make sure the lock is eventually released. The protocol is two phase because when there is a lock acquiring phase and a lock releasing phase.

Predicate locking

The only aspect of the implementation where predicate locking could be useful is in the Bid method. Locks could be placed on the items a particular client wants to bid on, so that multiple clients can bid on several items at once. However, it is more likely that clients are bidding on the same items,moreover, even if they are not, a client bids on a set of items, if we are to have predicate locking, this will lead to a lot of locking overhead that is not likely to match the added level of concurrency achieved.

Performance

I have used a pessimistic approach to implement concurrency. I make use of single write lock and a shared read lock, this keeps dirty reads, dirty writes and deadlocks at bay. The level of concurrency achieved is much higher than a system that implements a single lock for read and write, this is because here multiple reads can occur at once.Also, it has less locking overhead compared to a system that implements predicate locking. I assume that bidding will be the most frequent, interaction in the auction market; a system that will achieve higher throughput, should allow multiple clients bid at once while minimizing locking overhead.

3.3

I have proceeded testing by first checking the correctness of the operations of auction market then checking that appropriate exceptions are thrown when invalid input is given, lastly, I have tested how it behaves in regard to concurrency by spawning multiple threads.

All-or-nothing

testAddInvalidItemId(), this test attempts to add two items to the store, one with a valid item ID and the other with an invalid one. Both items are not added. The rest of the tests follow this same strategy in making sure an operation is not done half way; I have done this for invalid seller Ids, null input and invalid item description.

Before-or-after

I have tested that the auction market is not seen in an inconsistent state when an operation is being carried, this ensures that there are no dirty reads. In this implementation, it is not possible for dirty writes to occur because, only one write can happen at a time; none will overwrite the other.
testAddQueryConcurrency; T1 invokes add items while T2 is querying items consistently, it returns true if the market is either empty or contains the items added.
testQueryAndSwitchConcurrency; T1 invokes query items consistently while T2 invokes switch epoch, the market should either be empty (indicating the epoch has been changed) or contain the initial items, indicating otherwise. The rest of the tests follow this same strategy.

3.4

Setup

The experiment was carried out on a laptop with processor Intel(R) Celeron 1037U @ 1.80Hz with 2 cores, a RAM of 4GB, Before the experiments the CPU usage was about 34% and the memory usage was about 82%. The hardware of the system is not so great, so the speed of operations will not be as good as a system with 4 cores or more.Also,the level of memory available was a little bit inadequate. This experiment was averaged across different times to account for different levels of computer usage. The items and bids are generated randomly, at the beginning of the experiment 20 items are added to the market, so that bidding can take place. The item IDs are generated randomly, descriptions are appended with an integer (their value is not of much consequence to the experiment) and the seller id is also generated randomly. The items IDs for the bids are not so random because there has to be items in the market to bid on, consequently, random items are selected from the market to be bid on. The price and buyer organization ID are also generated randomly. The use of random numbers for item IDs is because duplicates are not allowed. For the NON-RPC, the experiment is carried out for 1-50 buyer clients, while in the RPC is is carried out from 1-20, this is because it took a sufficient amount of time to carry out one iteration.However, I have varied both long enough to see a pattern in the experiment which is my goal. For the workload, I have made add-items and query-items run 10

percent of the time, and Bid to run 70 percent of the time, I expect in a real auction market, bidding will be the most frequent operation. The throughput is calculated as $\sum_{i \in Workers} \frac{successfulFrequentInteractionsRuns_i}{elapsedTimed_i}$.
Before the actual experiment was carried out, I carried out a number of warm runs, so that the values calculated are not affected by the starting overhead. 100 is a good number, I have seen that numbers above 100 tend to slow the system down significantly.Generally, a higher number contributes to a clearer result, less starting overhead.The actual runs are done 200 times, this is adequate to get a stable result. The timing is calculated from the beginning of an interaction till the time it returns to the function that invoked it.

Results

Non-RPC

Figure 2 shows that the local throughput has its peak at about 6 threads, with nearly 10^5 operations per second but after this there is a steady fluctuation from 10 to 15 threads, from 15 threads upward the system thoughput begins to rise and fall drastically except between 30 and 40 threads where it is steady. These fluctuations could as a result of thrashing as increase in the number of threads no longer give an increase in the amount of throughput.This usually occurs as the number of blocking threads increases; since the most frequent interaction is the Bid; this is expected. However, the extreme rising and falling of the throughput could be as a result of not having enough warm-up runs or it could also be noise.

RPC

Figure 3 shows that the RPC throughput has its peak at 1 thread, which is much less than the non-RPC, however, considering the overhead of the communication link and the efficiency of my laptop; this is expected. The throughput drops steadily has the amount of threads are increased and it only increases slightly when we have 18 threads(this is most likely noise). In general, for both RPC and non-RPC , I expect that the throughput increases steadily as the number of threads increases but after a while it should begin to thrash; that point of thrashing is informative of the amount of workers that an admin client should spawn (after that point, spawning more threads becomes unnecessary).

Figure 2

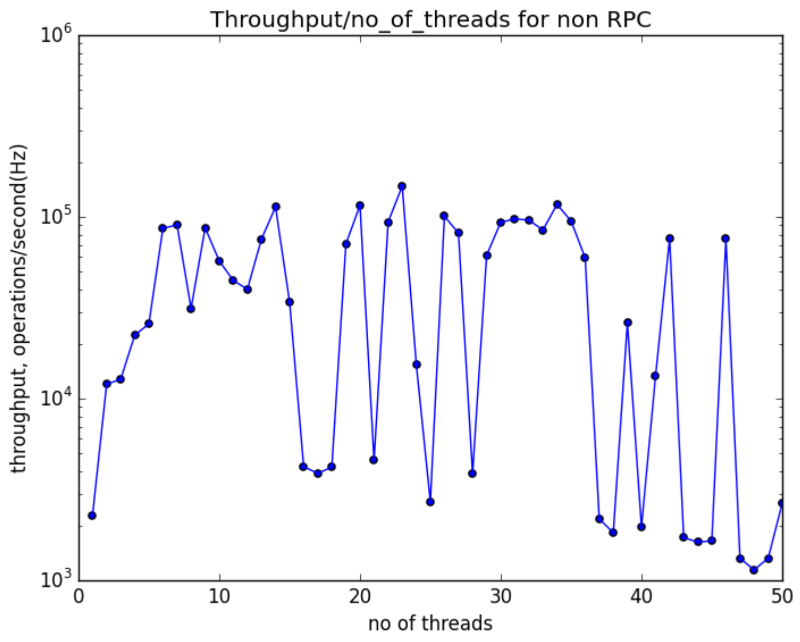
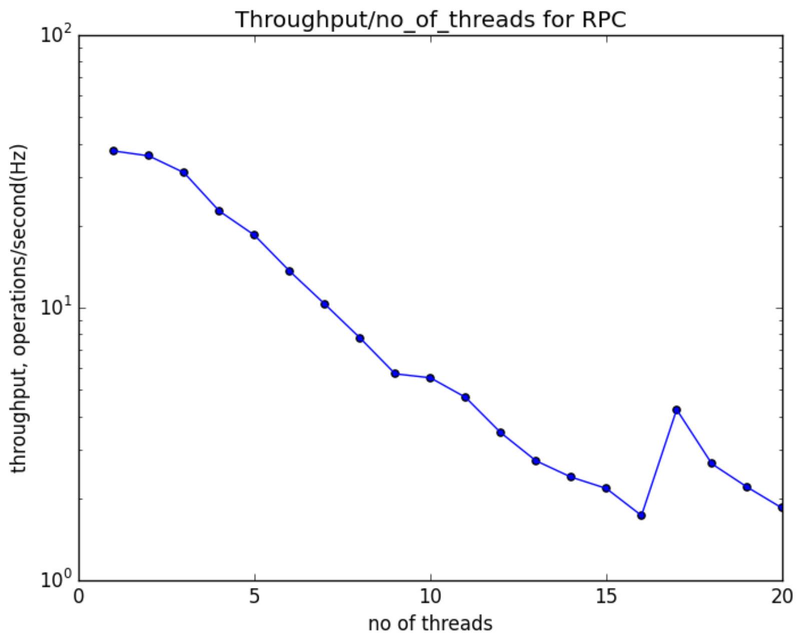

Throughput/no_of_threads for non RPC
throughput, operations/second(Hz)
no of threads
10^3
10^4
10^5
10^6
0
10
20
30
40
50

Figure 3


Throughput/no_of_threads for RPC
throughput, operations/second(Hz)
no of threads
10^2
10^1
10^0
0
5
10
15
20

References

[1] Department of Computer science,University of Copenhagen. *Advanced computer systems compendium*: 2015/2016.

[2] 8.6. External Sorting — CS3114 Fall 2015
`http://algoviz.org/OpenDSA/Books/CS3114/html/ExternalSort.html`