

Principles of Computer System Design

Exam

Tudor Dragan (xlq880)

January 22, 2015

Question 1: Data Processing

1. Sort-based external memory algorithm

For this part of the exam we must implement an algorithm that first applies an aggregated function (in our case the *count* function) on the table *friends*, and then combine the results to write up the table that consists of (uid, network, nrOfFriends). In order to achieve this, I used a modified sort-merge join algorithm.

I made the following assumptions:

1. We don't have any indices on the tables so the entries are not in sorted order in any of the two tables.
2. The *friends* table has only uni-directional relations, in the sense that *Person1* can be friends with *Person2* but it's not required that *Person2* has to be friends with *Person1*. (*Person1* has 1 friend and *Person2* has 0 friends.)
3. The table with the biggest number of records has N records.
4. We know that the main memory can hold \sqrt{N} records. If we read B records at a time in memory, the number of runs will be N/B .
5. Number of passes in Phase 2 is P then: $B(B-1)^P = N$

First I build the table with the aggregate count function for the *friends* table by altering the way that we merge the pages on a pass. I will explain how it is done in the following paragraphs on a concrete example:

1. Every entry in the *friends* table has this form : (*uid1*, *uid2*). The input is split up in multiple blocks. Let us assume that we have a block that has these following (*uid1*, *uid2*) relations: (3,7) (1,4) (2,3) (2,5) (1,3) (1,2) (2,4) (3,1)
2. In order to calculate the number of friends for each user, I will sort records with the form (*uid1*, *friendCount*). because every record in the *friends* table essentially means 1 friend added, when reading in the first phase, the *friendCount* is 1. So we will have initially tuples with the form (*uid*, 1) that need to be sorted by *uid*. We assume that the buffer

- page number is 4. So after the first phase we will have [(3,1) (1,1) (2,1) (2,1)] [(1,1) (1,1) (2,1) (3,1)].
3. The next pass will be [(1,1) (2,2) (3,1)] [(1,2) (2,1) (3,1)], Then we combine the values by adding up the number of friends by comparing the heads of the lists (the heads contain the smallest *uid*) and add up or write to disk the smallest *uid* with the friend count.
 4. Finally we will have [(1,3) (2,3) (3,2)].
 5. This will continue until all the pages buffers are empty and have no more data to fill them up with.

After we create the number of *friends* table that has entries sorted by *uid*, we sort the *user* table. We sort the users by *uid* because we would like to split both sorted inputs into blocks and combine the result to (*uid*, *network*, *numberOfFriends*) output form. Because we have space in memory for \sqrt{N} records, each buffer block should have a size of $\sqrt{N}/2$. We compare the heads of the lists, because they always have the smallest possible *id* and merge the values together. We continue filling up the buffers and stop when there are no more entries to compare.

2. Hash-based external memory algorithm

The hash-based algorithm is very similar to the *GRACE hash join algorithm* with a slight modification that increments a counter for displaying the number of friends for each user. We make the following assumptions:

1. There are $B - 1$ buckets, with B main memory buffers. These buckets will hold $\sqrt{N}/2$ records, where N is the highest of U and F .
2. The hashtable in the second phase is smaller than $B - 1$ pages.

We first partition both relations U (user table) and F (friend table) via a hash function into $B - 1$ buckets. We read one partition at a time. We iterate through it and place the record in one of the buckets by the hashing the join attribute (in our case the *uid*). Then we can be sure that if tuples of U and F join, they will wind up in corresponding buckets U_i and F_i for some i . After this phase, we read a bucket of U from disk and hash it using another function and construct the hash table in memory. Then we load in a buffer for S_i and iterate through it. When we match the *uid*, we increment the *numberOfFriends* counter in the hash table. When the buffer is empty, we write to disk the hash table with (*uid*, *network*, *numberOfFriends*) and load the next entries in the buffer.

3. I/O costs

Sort-merge algorithm

For the sort-merge algorithm, the cost is:

$$Sort_U + Sort_F + 2 * Pages_U \quad (1)$$

where $Sort_U$ and $Sort_F$ = the cost of one external merge sort on the user table and friend table respectively with:

$$Sort_U = 2 * U(\log_2 U + 1) \quad (2)$$

$$Sort_F = 2 * F(\log_2 F + 1) \quad (3)$$

and $Pages_U$ and $Pages_F$ = the cost of page writes/reads (concurrently).

After the sorting of the friends table, the friend table will have a lower or equal amount of pages as the user table because we aggregate the results by counting up the relationships between users. Because of this instead of having $Pages_U + Pages_F$ we would have in this scenario $2 * Pages_U$. The result would have the same number of pages as the users table.

Hash-based algorithm

The hash-based algorithm has a cost of:

$$3 * Pages_U + 2 * Pages_F \quad (4)$$

where $Pages_U$ and $Pages_F$ = the cost of page writes/reads (concurrently).

This total cost reflects the fact that after the split in buckets and combining them using the GRACE hash join algorithm, we only construct on disk a final table that has the same number of pages as the user table, so we would have $2 * Pages_F$ instead of $3 * Pages_F$.

Question 2: Distributed Transaction

1. Local wait-for-graphs

Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

A schedule is conflict-serializable if and only if its precedence graph has no cycles. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions. The local wait-for graphs on each node are:

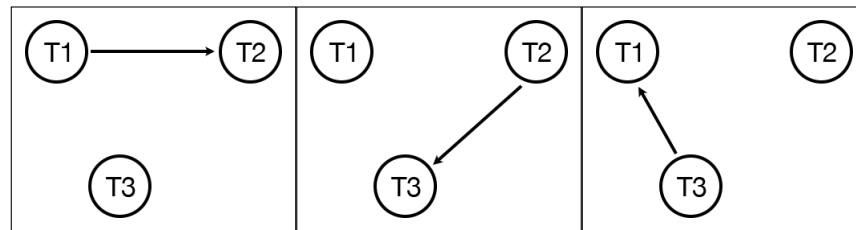


Figure 1: Local wait-for-graph for Node 1, Node 2 and Node 3

As we can see the graphs have no cycles, proving that there are no local deadlocks.

2. Global wait-for-graph

When constructing the global wait-for-graph we set up the pages that are being modified on each machine. If we have a cycle in the global graph, it means that we have conflicts and must abort the transactions. Below we have the global wait-for-graph:

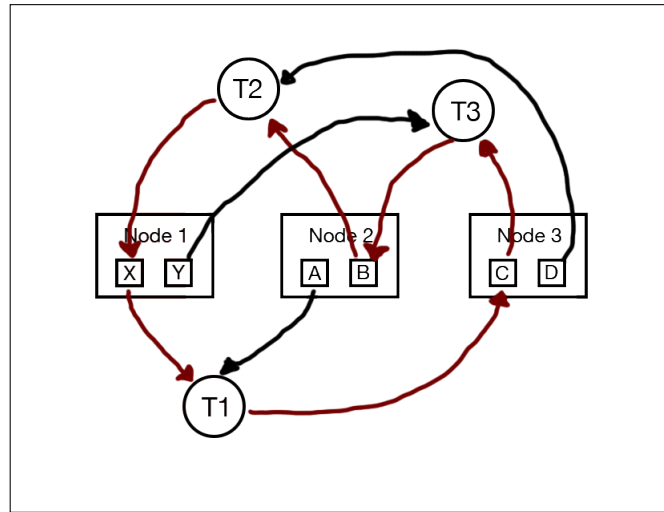


Figure 2: Global wait-for-graph

As we can see, we have cycle between $X \rightarrow T1 \rightarrow C \rightarrow T3 \rightarrow B \rightarrow T2 \rightarrow X$. To remove the deadlock within the system, we must abort modifications that are being done on X , C and B .

Programming Task

Question 1

For the programming task I emphasized on building a strong, modular service that is based on RPCs. In the following paragraphs I will discuss on the main components of the *acertain-farm.com* web-service. As we can see in Figure 5, the application is split up into 2 different servers: the *Sensor Aggregator Server* and the *Field Status Server*. Both of these servers are implemented by instantiating a multi-threaded server using the Jetty framework¹. The methods available for both servers are split up into two interfaces: *SensorAggregator* and *Field Status*. These interfaces are implemented by *proxies* that serialize and send out the requests for the server to handle. The *Sensor Aggregator Server* implements the *SensorAggregator* interface in the *FarmSensorAggregator* class whose methods are invoked by the handler class *SensorAggregatorHTTPMessageHandler*, which handles the request that were received by the Jetty HTTP server class *FarmSensorAggregatorServer* class. A similar approach is done on the *Field Status*

¹The Jetty 8 library (<http://www.eclipse.org/jetty/>) is used to provide the HTTP server and the HTTP client libraries.

Server, but using another interface (the *Field Status* interface).

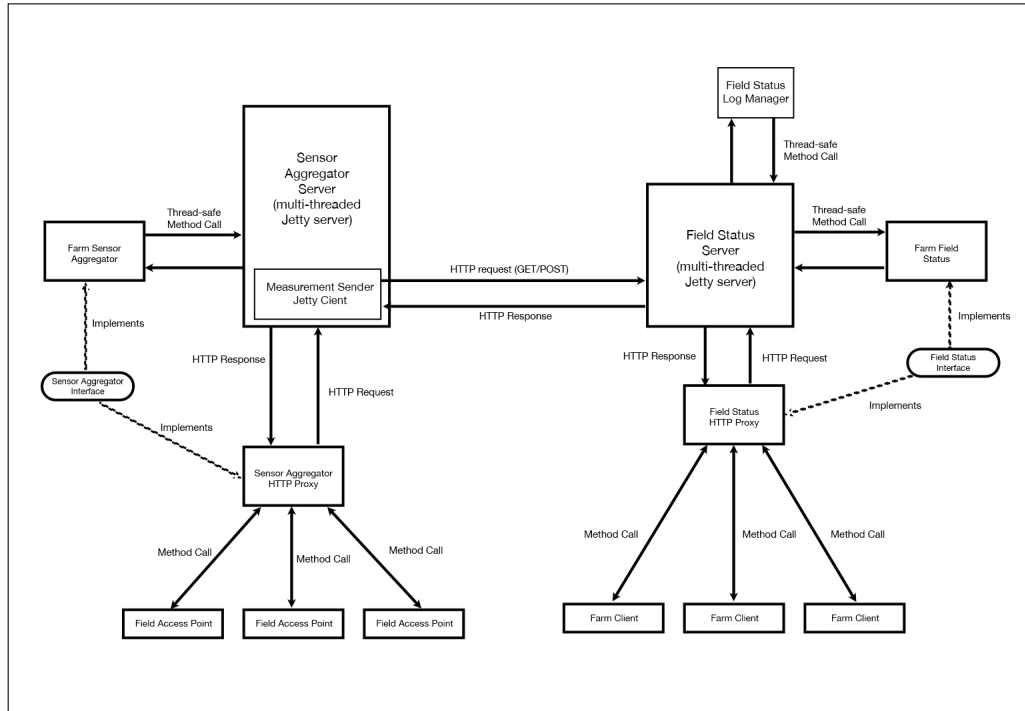


Figure 3: Architecture of the farm's monitoring application

The RPCs are done by using a Jetty HTTPClient, which sends out HTTP request for the server to handle. This *exactly-once RPC mechanism* is present through the application and is used for communicating between the *FarmFieldAccessPoints* and *FarmSensorAggregator*, *FarmMeasurementSender* and *FieldStatusServer*, and *FarmClient* and *FieldStatusServer* by keeping track of requests and responses. In the case of the *Sensor Aggregator*, the *Field Access Point* sends out a xml-serialized HTTP request through the *FarmSensorAggregatorHTTPProxy* class. The request is sent and handled by the server. When the server receives the requests, it deserializes it in the handler and passes it to the *FarmSensorAggregator* instance through a thread-safe method call. Then the request is processed by the *FarmSensorAggregator* class. between the *Sensor Aggregator Server* and *Field Status Server*, the communication is done directly by using an HTTP Client instantiated on the server (in our case, it is encapsulated in the *FarmSensorAggregatorSender*) that sends out POST Requests directly to the *Field Status Server* (the request doesn't pass through the HTTP Proxy).

Because I use two different servers, each one is independent from another. In a more concrete case, if the *Field Status Server* is not responding the *Sensor Aggregator Server* continues polling and processing *Field Access Points Measurements*, but the requests that are being sent to the *Field Status Server* will respond with a TIMEOUT by the *FarmSensorAggregatorSender* class that conforms to the fail-stop model.

On other side, in the case of the *Sensor Aggregator Server* failure, the *Field Status Server* can continue handling requests from *Farm Clients* with the last known measurements that it received from the other server. As we can see both failures are contained within each service and one does not influence the behavior of the other. The *Field Status Server* continuously listens for further updates from the *Sensor Aggregator Server*.

Question 2

For the implementation of the locking protocol used in the *FarmSensorAggregator* class I used the *ReentrantReadWriteLock* class available in the *java.util.concurrent.locks* package. A *ReentrantLock* is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This

Lock type	read-lock	write-lock
read-lock		X
write-lock	X	X

Table 1: Lock compatibility table

approach guarantees atomicity within transactions but does not allow a lot of concurrent access to the map that is modified. I provided an all-or-nothing approach by aborting the entire transaction, when an input is invalid. I implemented the strict two-phase locking (S2PL) by requiring a write lock when modifying the data at the start of the method.

The exam question states that it is recommended that we do a fast path solution. First I tried using a *Multiple Granularity Lock Protocol* because we need to modify only a key-value pair in the map and we do not want to keep a lock on the entire *HashMap*, so that other threads can modify different pairs concurrently. I tried using the *ConcurrentHashMap*² class that is part of the *java.util.concurrent* package, but eventually I gave up on the implementation and went with the safe approach. I had problems when sending the map to the *Field Status Server* through the *Sender* class so I decided it was best to stick to the solution that works, rather than going with one that didn't.

Question 3

For ensuring serializability of transactions on the *Field Status Server*, I keep track of the last time period that the *Sensor Aggregator* sends. By comparing the two values upon the arrival an update request, I check that the timestamp is newer than the one currently present in the *HashMap*. After each update request I require an write lock using the same locking system as in

²A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as *Hashtable*, and includes versions of methods corresponding to each method of *Hashtable*. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with *Hashtable* in programs that rely on its thread safety but not on its synchronization details.

the Sensor Aggregator. If the timestamp is pointing out an earlier time, the request is discarded.

Because update request are not as frequent as query operations, it made sense that i would use a conservative locking protocol for ensuring serializability. The serializability of the transaction (update and query) are done by a simple locking mechanism. When an update arrives, I acquire a lock on the data store, which means that the query operations will have to wait for the release of the write lock. If there are only queries, the operations can occur concurrently.

I used predicate lock³ for ensuring that the updates and queries are atomic operations. The idea is to acquire a shared lock on the QUERY predicate and an exclusive lock on the UPDATE. The lock can't be granted if a conflicting lock is held already. This maintains the ACID properties and ensures serializability between calls.

We could improve our current solution by adding an optimistic approach on how the locking is done. Instead of locking the entire HashMap we could lock the objects at a key accessing level. This is done by using the following code:

```
class RWDictionary {
    private final Map<String, Data> m = new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public Data get(String key) {
        r.lock();
        try { return m.get(key); }
        finally { r.unlock(); }
    }
    public String[] allKeys() {
        r.lock();
        try { return m.keySet().toArray(); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

Figure 4: Improving concurrency by acquiring locks at the key level

³K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. Appears in Communications of the ACM, Vol. 19, No. 11, 1976

Question 4

Durability is achieved by writing the updates to a log. This is implemented by the *Field-StatusLogManager* class which access the log file for reading or writing. I implemented the *Write-Ahead Logging* protocol by forcing for an update to a log, before modifying the map. I wait for the log manager to finish writing to disk. This approach guarantees atomicity and durability. After each flush to the log I add a CHECKPOINT record so that in case of a possible failure of the *Field Status Server* we can REDO the operations from the CHECKPOINT entry. Because we are storing everything in main memory, we do not need UNDO events. We should however keep the data that comes through the update method, in order to REDO them. We write a full snapshot of the HashMap to the log, to rebuild the map in case of a possible failure, and the newly updated fields list.

In case of a system failure, we would need to REDO the events by reading from the last CHECKPOINT. Because we reside our data in main memory, we do not need to UNDO other updates. In the log I first write the SNAPSHOT of the map present in main memory, then I write the UPDATE records that came with the update request from the *Sensor Aggregator Server*. By implementing this mechanism of log keeping we can reconstruct the HashMap during the recovery phase.

Because the two servers are independent from one another, we would not need to apply any special actions to start them up. The *Sensor Aggregator* continues to send updates to the address specified and if the *Field Status Server* is unreachable, it would get a TIMEOUT response (this complies with the fail-stop model).

Question 5

For testing I used a combination of manual and automated tests. The automated tests are implemented with *JUnit*. For the functionality of the services I did white-box testing that test the internal structures of my application. For testing the proxy interface I used black-box testing to simulate how a real client would actual use the service.

Error conditions are met and the proper exceptions are being thrown on each step. This ensures that we have consistent data across the system so that we don't have anomalies in calculations that may output wrong values. I have written JUnit test classes for testing each component in the system. The *WorkloadTest* class test the system as a whole by enabling all components.

Question 6

The setup that I have chosen for the workload tests is the following:

The workload test were run by emulating the two Jetty servers and creating a workload test that is similar to the *CertainBookStore* implementation. The *FarmWorkloadConfig* contains parameters that can be modified for testing different scenarios. By default I have 10 *Fields* and 10 *FarmAccessPoints* that output 10 random *Measurements* each for the *FarmSensorAggregator*. The *FarmAccess Points* output *Measurements* for different fieldIDs. The *Measurements*

Model	Macbook Pro 15" (Early 2011)
CPU	2 GHz Intel Core i7 (8 threads)
Memory	8 GB
JRE	Oracle JRE 1.8

Table 2: Hardware setup

are sent to the *SensorAggregator* every 500 milliseconds by each access point client. The *SensorAggregator* outputs every 5 seconds the aggregate of the measurements for each field. The following graphs represents the throughput and latency for a workload test that performs 1000 queries for 8 fields. The throughput is calculated as in the *CertainBookStore* assignment and provides us with the following values for multiple worker threads.

We observe that the service throughput does scale, but not perfectly. My tests simulate between 1 and 10 *Farm Clients* that interact with the *FarmFieldStatusServer*. We can observe an increase in throughput when adding more *Farm Clients* but the highest throughput value is at the 8 thread mark. This makes senses because the system on which i run the experiments has only 8 threads. From the latency graph we see a steady increase by adding more requests to the service.

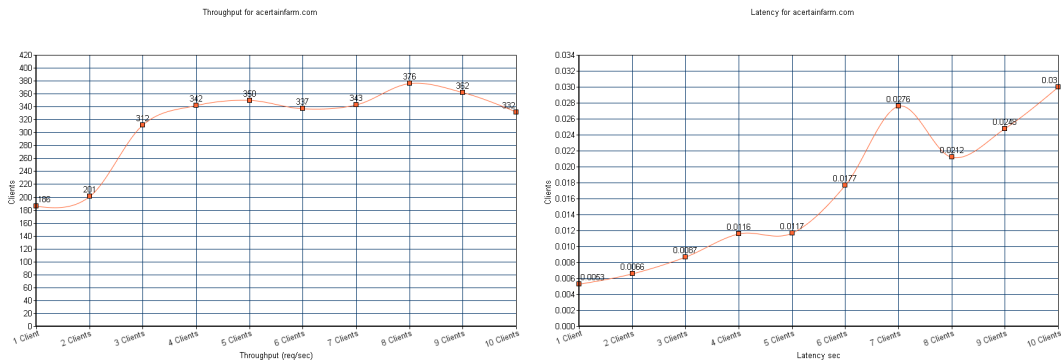


Figure 5: Throughput and latency