Advanced Systems Lab (Fall'15) – First Milestone

Name: *Sandro Huber*
Legi number: *10-924-777*

Grading

Section	**Points**
1.1	
1.2	
1.3	
2.1	
2.2	
2.3	
3.1	
3.2	
3.3	
3.4	
3.5	
3.6	
Total	

1 System Description

1.1 Database

Length: 1-2 pages

Start by explaining the schema of the database and the indexes used to speed up data access. Describe the interface to the database (queries and stored procedures).

Make sure to explain the design in terms of what you wanted to achieve, what decisions you took and what is the expected behavior.

Include baseline performance characteristics of the database (max throughput, response time, and scalability).

1.1.1 Schema and Indexes

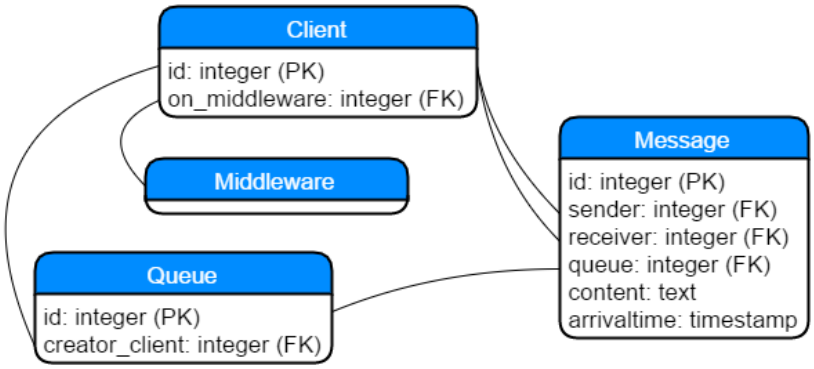


Figure 1: The Database Schema

The database schema was chosen in such a way, that it provides all required functionality, but is still easy to setup and maintain. The following lines shortly illuminate the tables and their column types.

- **Client**: Guarantees through the serial datatype that each client in the whole system has a unique ID and stores which client is registered on which middleware. This can be handy when wanting an even distribution of clients over all middlewares.
- **Queue**: Stores all queues created by clients. Also ensures that all queues are unique.
- **Middleware**: A sequence which is used to get a unique ID for each middleware joining the system.
- **Message**: Here, all messages sent by clients are stored. The foreign keys ensure that we only get messages from valid clients registered in the system. As one can see, the content column is of type text, despite the choice of varchar(200) or varchar(2000) may be more obvious with respect to the system requirements. Find my thoughts about this choice in section 1.1.3.

To speed up the data access I used the two following indices:

```
(a) INDEX msg_rcvr_q_idx ON MESSAGE (RECEIVER, QUEUE);
(b) CREATE INDEX msg_sndr_idx ON MESSAGE (SENDER);
```

The reasoning behind the choice of them is based on the following thoughts. The indices in a database system affect search time, i.e. filtering of rows with respect to a criterion. Since the whole system is built with predefined stored procedures (find a full list in section 1.1.2) we know exactly what these filter parameters will be. To simplify the view, only the most relevant parts of the queries are shown (query they belong to in brackets):

(1) **WHERE** RECEIVER (get_queues_for_client)
(2) **WHERE** RECEIVER **AND** QUEUE (read_all_messages_of_queue)
(3) **WHERE** RECEIVER **AND** QUEUE **ORDER BY** ARRIVALTIME (remove_top_message_from_queue)
(4) **WHERE** RECEIVER **AND** SENDER **ORDER BY** ARRIVALTIME (read_message_from_sender)

Since an index on two columns (c1, c2) is also an index onto c1, we can see that the index (a) already covers the WHERE clauses of (1)-(3). In addition with index (b) we get a full coverage of all queries having to filter some data. It is intentional that there is no index on the GROUP BY of ARRIVALTIME. Find more about the reasoning in section 1.1.3.

1.1.2 Stored Procedures

Every database access is done via a stored procedure. This allows to have a single point of failure, maintenance and control over functionality. Having this lone entry guarantees fast and reliable feature implementation and debugging. In Java the stored procedures are interfaced via Prepared Statments, which can be cached by the VM, such that only the dynamic parameter values have to get fetched, before a query can be executed. The following stored procedures are implemented:

```
create_queue(creator_client INTEGER)
delete_queue(queue_id INTEGER)
register_client(on_middleware INTEGER)
get_queues_for_client(client_receiver INTEGER)
read_all_messages_of_queue(receiver_id INTEGER, queue_id INTEGER)
read_message_from_sender(sender_id INTEGER, receiver_id INTEGER)
remove_top_message_from_queue(receiver_id INTEGER, queue_id INTEGER)
send_message(sender_id INTEGER,
        receiver_id INTEGER, queue_id INTEGER, content TEXT)
register_middleware()
get_registered_clients()
get_number_of_messages()
get_registered_queues()
take_stamp()
```

1.1.3 Design decisions

The design aims to be simple, but yet complex enough to provide all necessary functionalities through a nice interface. Please find in the following lines the reasoning about some design decision I took while implementing the system:

- **text vs varchar(n)**: First of all, all datatypes char(n), varchar(n), varchar and text are internally all converted to the same C data structure varlena. So from the performance perspective no (major) differences are measurable. Because I wanted to be flexible and send messages of length 200 and 2000 in the same setup I decided to go with the flexible variant text. Choosing varchar(2000) for every case is bad, because the unused space when inserting smaller strings will be filled with spaces and thus not give any performance advantages.

- **Ghost-Client**: Since messages can also be addressed to all clients in the system, I introduced a ghost-client with an ID of 0, which gets created right at the database initialization. This ghost-client allows that instead of having RECEIVER=NULL, we have RECEIVER=0 for all broadcast messages, which is internally much easier to handle.

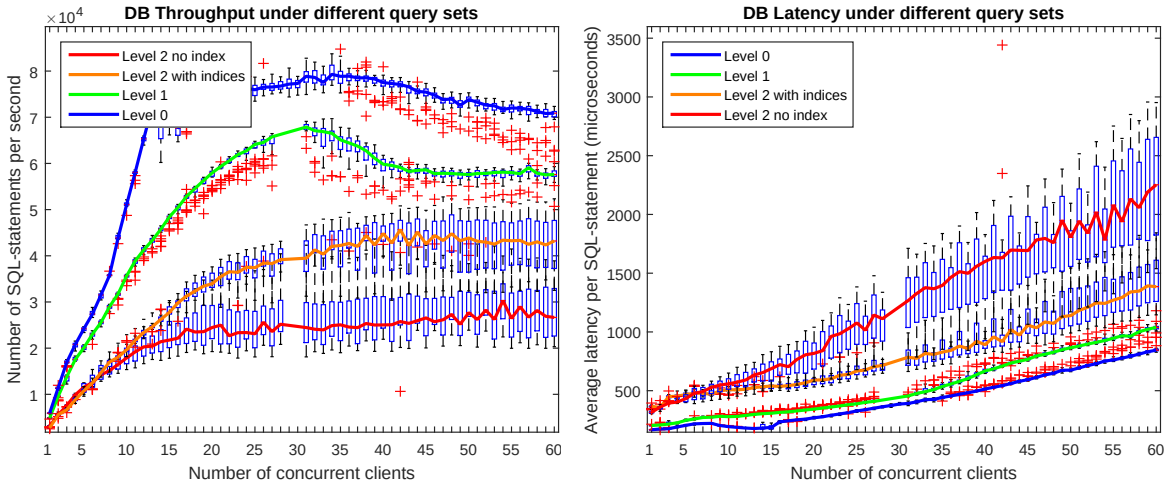
- **Index on ARRIVALTIME**: Maintaining an index is not cheap for a database system, so it's wise to use them with caution. Because of that I decided that the indices (1) and

(2) are already discarding enough rows, such that the sorting operation is not too costly anymore.

- **ARRIVALTIME location**: The ARRIVALTIME is implemented as a trigger on the database. Another, also valid choice, would have been to let the client set it. But because I wanted to minimize the work of the clients and guarantee a unique timestamp on each message I did choose the trigger-option.

1.1.4 Performance characteristics

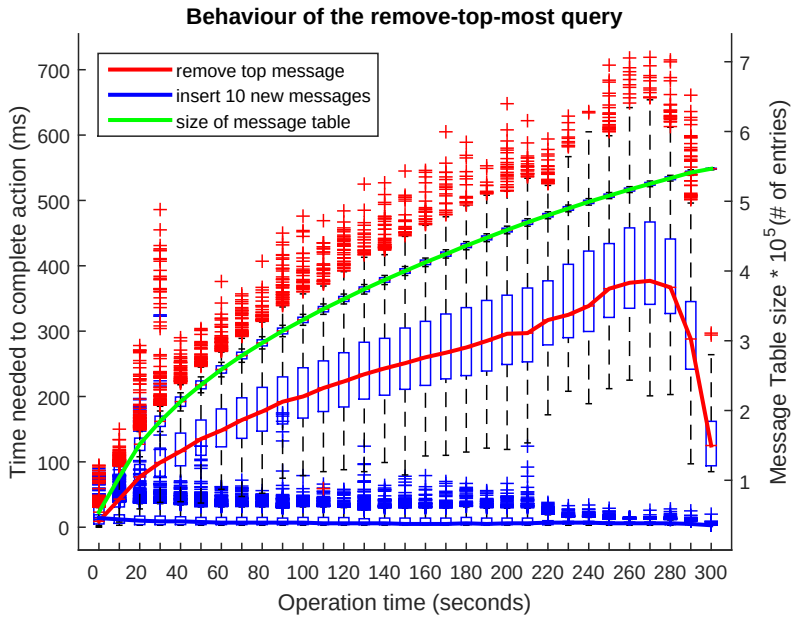
To measure the performance of the database I used the postgresql tool pgbench. For the machine details, please refer to section 2.1. I defined three levels of difficulty, which correspond to different query sets, level 0 beeing the easiest, level 2 the hardest. This allows us to get a smooth transition from absolute best to real-world behaviour. For a detailed insight into the levels, please find them in /db_baseline/benchScripts. All contents had length 200. Before running the presented experiment I tested how many clients per database connection are optimal. I expected that the relation of 1:1 should hold, which indeed was the case. For the throughput and latency I expect the knee to be found around 16 clients, because we work with a machine that provides 8 physical cores and thus 16 hyperthreads. Of course the throughput level ranking from hightest to lowest should be: 0, 1, 2+, 2- (+/- for with or without indices), and vice-versa with the latency. After the knee there should at least be a visible stagnation of the throughput, but a remarkable increase of latency.



Firstly please note, that the lack of data around 30 clients is explained through a bug in the code. But it does not seem that any exceptional behaviour is missed due to that. As one can see (the lines follow the 50% quantiles), the knees can really be found around the expected 16 clients. Also the rank of the levels behave as expected. I am especially happy about the performance benefit gained by the indices. Only the gain of latency after the barely visible knee is not as steep as foreseen. But this is probably okay, since we can't see a total throughput breakdown neither. The most costly operation in the whole benchmark is identified as the removal of the top message, i.e. executing *remove_top_message_from_queue*. To further investigate this I performed the subsequent follow-up experiment.

The speed of the top most message removal is highly correlated with the size of the database, because the more messages, the longer the index access and the longer the sorting (ORDER BY) of the final table entries will take. To simulate an extreme scenario I only registered one client and one queue in the system into which all messages were sent. All messages also had

sender and receiver set to the lone client and had a random content of length 200. This allows to focus on the behaviour of the performance of the query with respect to the size of the database. The database was benched by 40 concurrent clients (for machine type reference, please look up section 2.1), operating on 40 database connections. I chose 40, because we just saw, that around this many clients, the database throughput on the level 2+ was maximal. Each client ran independantly and chose randomly between either inserting 10 new messages, or removing the top most message of the single queue 3000 times. This workload ensured, that the database grows over time and the whole experiment runs sufficiently long. According to the behaviour in the last experiment I expect a steady increase of the latency, following the trend of the database size quite closely. It's clearly visible that the expectation of having a close relation between



the performance of the query and the database size is true. To be sure, that the insertion of new messages does not blur the picture, also this data is plotted. The abrupt break-down of the latency at the end of the experiment is due to the completion of the first few clients. This allows the ones still running to get a congestion-free access to the database. It is visible, that the database stays stable, even when going up to half a million entries in the message table.

1.2 Middleware

Length: 1-2 pages

Explain the design from a high-level point of view, highlighting what you wanted to achieve, design decisions, expected behavior.

Then go into more detail on how the middleware connects to the database and clients, and how queuing is implemented.

Show what are the performance characteristics of the middleware (i.e. throughput, latency, scalability).

1.2.1 Design overview

1.2.2 Interfacing with clients

1.2.3 Queuing and Connection pool to database

1.2.4 Performance characteristics

1.3 Clients

Length: 2-3 pages

Explain the interface of the clients to your messaging system and their high level design, including the ways you have instrumented the code for debugging and benchmarking purposes.

Provide a detailed description of the workloads used later in the report (operation mix, starting and ending state of the database, assumptions on workload behavior). Explain how the load was generated (include baselines on load generation speed) and how the clients were deployed.

Which are the sanity checks in place for ensuring correct load generation and validity of responses?

1.3.1 Design and interface

1.3.2 Instrumentation

1.3.3 Workloads and deployment

1.3.4 Sanity checks

2 Experimental Setup

Length: 1-2 pages

Explain the overall design of the complete system and list the configurations (number of middlewares, number of clients, types of machines, communication patterns) corresponding to the main workloads.

Describe the mechanisms for deploying the system for experiments and the way performance numbers are gathered and processed. Make the description so that someone unfamiliar with your system can replicate the steps, and reference the different script files you submit as code in the SVN repository.

2.1 System Configurations

2.2 Configuration and Deployment mechanisms

2.3 Logging and Benchmarking mechanisms

3 Evaluation

Length: up to 10 pages

In this section we expect to see the different experiments you ran to exercise the system, and with each experiment we expect a clear description of the system configuration used, the hypothesis on behavior and the explanation of the behavior observed (in terms of the different design decisions taken beforehand) – *missing either of these for an experiment might make you lose all points for that given experiment!* Keep in mind that for a good explanation of the results of an experiment you might have to use one or more methods of data analysis presented in the lecture and in the book.

See below for a short description on what each part should contain.

3.1 System Stability

To prove that your system functions correctly and that it is stable include the trace of a 30 minute run, plotting both response time and throughput. Use at least 30 clients (sending and receiving data), 2 middlewares and a non-empty database.

3.2 System Throughput

Measure the maximum throughput of the system (describe the exact configuration and workload, and the reasoning behind choosing these particular ones) and show the average response time for this experiment.

3.3 System Scalability

Explain the different configurations used to explore the scalability of your system, and the outcomes of these experiments in terms of throughput and response times. The main goal of this subsection is to define the ranges in which your system operates best.

3.4 Response Time Variations

Report and analyze how the response times change in the system with different message sizes, different number of clients and different number of middleware nodes.

3.5 2^k Experiment

Conduct a 2^k analysis of your system (aim at exploring non-obvious interactions of parameters). Use the methods learned in this lecture to conduct the detailed analysis.

3.6 Conclusion

To conclude the report summarize the behavior of the system in terms of the design and the representative workloads. Finally, outline in a few points what would you do differently if you could design the system anew.