1. Designing the timeseries storage system.

There is a set of high-frequency sensors (up to 1000 sensors, each 1 kHz) that generate data every second. We can assume that the data is sent to storage by already formed batches coming from some intermediate system that has the ability for very short-term storage (up to 3-4 seconds), and is also unreliable. It should be noted that due to the imperfection of the device of the sensors themselves and the specified intermediate system, some data may be significantly delayed.

The generated data must be written to some storage/database for permanent storage. Requests will be made to this storage to ensure the operation of the “lenses” system (data lenses) associated with the visualization of aggregated time series for the user interface. Each of the lenses involves viewing a certain time interval selected by the user (for example, 10 seconds, 1 minute, or 1 hour), specially aggregated into such a number of points that can be displayed on the screen (for example, the interval of 1 minute for a 1 kHz sensor is 60,000 points, which are aggregated (roll up) into 2500 points by finding the average for each 60000 / 2500 = 24 consecutive points). A user can request from 1 to 10 sensors with the same time interval within a single request, and several users (up to a hundred) can work with the storage at once. Storage conditions mean that there may be short-term failures of nodes where the storage is deployed or networks before them (up to 1 min.).

You need to design a storage together with the data loading system (ingestion), if necessary, that would meet the following functional conditions:

- Queries must be correct, i.e. they must always return the value that would have been obtained if the “raw” data was accessed directly.
- Fast data acquisition, i.e. there should not be a scan of “raw” data with their full aggregation.
- It is advisable to avoid writing to the storage too often.
- Small data loss during recording is allowed.*

*If you are able to explain the compromise.
There are several variants related to data losing:

1. No data losing in raw data + No data losing in lenses (in this case you will get extra points).
2. No data losing in raw data + Some data is lost in lenses.
3. Some data is lost in raw data + Some data is lost in lenses.

Perfect case (additional condition):

1. There are no duplicate records in raw data and lenses (in this case you will also get extra points).

Requirements:

1. Use ClickHouse as a main storage.
2. Sharding and replication (at least 4 servers).
3. Data volume being stored should contain at least one month of data (quantity of sensors and their frequency see earlier in the text).

4. Data generator(-s) and ingestion procedure is a must (note: one may use Kafka to reliably insert data).
5. Generators of request workload should be implemented. The storage should be able to insert and process select queries simultaneously.
6. Test scenarios should be implemented as a demo of the solution.
7. **Bare-metal deployment is forbidden. Deploy the storage with replicas, queues, generators, requesters and other modules using Kubernetes, docker (see: Kubernetes' StatefulSet / Deployment / Job, etc.)**

2. Logs, metrics and business processes.

Imagine we have a web platform for online sales. Users may visit and view different pages on the platform's website and perform various operations, mainly: reserving, buying, refunding, canceling reservations. The platform is implemented as a fleet of micro services and to perform financial transactions may communicate with external services (thus they are out of our control) depending on the user's prefered way and provider to make payment. Multiple micro-services participate in processing every operation thus it is guaranteed that each has a unique ID associated with it, staying preserved across all microservices. Users may make mistakes when they reserve or buy, external services may be temporarily unavailable or overloaded to process payment requests. The platform's software and the external services also may contain bugs that lead to various errors.

This fleet constantly generates logs, metrics and database records. The business wants these logs and metrics to be collected to calculate various statistics and to check for performance problems, errors and anomalies. You need to build a system that may serve analytical queries, raise alerts when anomalies are detected and fill dashboards dedicated for monitoring (not realtime, but close to it).

Incoming data of the last month should be stored in a "hot" storage for fast operative access while older data should be moved to the archive on S3. Alerts should be raised either instantly or in a matter of a few minutes if the event is complex (for example, multiple unsuccessful retries for several users with the same payment provider).

To make your task a little less difficult, there already exists an infrastructure for data collecting from above-mentioned sources that writes collected data into a message queue system. Thus you only need to integrate with this MQ, not with the whole fleet.

Examples of possible user queries that needs to be satisfied:

1. Get all events across different microservices related to the same action ID (e.g. request tracing).
2. Get events by specific event level (debug/info/warn/error) and their type of one or more microservices for a specific interval.
3. Number of simultaneous users viewing / buying
4. Business metrics (for financial reporting and anomalies detecting to raise alarms):
   - transactions count per interval (15 minutes / 1 hour / 3 hour / 1 day)
   - page views per interval (15 minutes / 1 hour / 3 hour / 1 day)
   - retries with successful or unsuccessful outcomes (eventual ones)
5. Performance:
   - operation execution time (min, max, average, percentiles);
   - CPU / RAM consumption per interval simultaneously with number of users
6. Anomalies:

- multitude of same operations;
- buying and refunding, etc.

System requirements:

1. Sharding (at least 3 servers).
2. Replication.
3. Reading from a message queue system (Kafka, Redis).
4. Incoming data should be processed with a streaming solution (Spark Streaming, but an alternative may be proposed) to raise alarms if anomalies are found (applicable only for small periods of time: tens of seconds or few minutes).
5. Archiving is a must.
6. At least one dashboard is a must.
7. Data generator(-s) and ingestion procedure is a must (note: one can use libraries to fake logs, for instance https://github.com/mingrammer/flog).
8. Generators of request workload should be implemented. The storage should be able to insert and process select queries simultaneously.
9. Test scenarios should be implemented as a demo of the solution.
10. **Bare-metal deployment is forbidden. Deploy the storage with replicas, queues, generators, requesters and other modules using Kubernetes, docker (see: Kubernetes' StatefulSet / Deployment / Job, etc.)**

Extra bonuses:

1. Implement detecting anomalies that don't fit into the streaming window (for unsuccessful operations).
2. Use Apache Flink instead of Spark Streaming (Apache Flink is not presented in the course materials, so one needs to get familiar with it by himself/herself).
3. TBD

3. Graphs Warehouse & Processing System.

There is data scrapped from social network vk.com.
You have to design the warehouse to store and process this data. The warehouse is oriented on processing the entities with a large number of interconnections. Example of such an entity - user. Users have a number of friends, posts, subscribes, etc. Moreover, their friends also have friends, likes etc. It may be represented as a huge graph. Traditional RDBMS cannot handle a large data scale, KV and OLAP also are not suitable solutions for graphs.
On the other hand, your warehouse has to provide the ability to work with some entities as documents (posts, comments).
Thus you have to use a multi-model solution.

Your warehouse has to be tuned and provide excellent performance for queries such as (examples):

- "Find all posts of friends of people who have liked some posts"
- "Find the shortest 'handshake path' between some users"
- "Find the comments of posts, that have been liked by user relatives"
- "Return all users' friends and friends of friends whose posts have been updated in period of between <date1> and <date2>"

You have to consider the data arrives constantly via message queues (messages are NOT ordered by scrap timestamp). Some messages are new entities, some messages are updates to the entity. But there are also possible duplicates.
To generate the data, use one of fake data generators. Data samples will be provided.

Also, you have to implement a monitoring system with dashboards for the storage. The monitoring system has to collect and store the data about the warehouse, sidecar services and node state. For example: CPU usage, RAM usage, disk usage. Number of requests (select/insert at all/per second), number of entities/records, etc.

Requirements:

1. Sharding and replication (at least 3 servers).
2. No data duplication.
3. No data losing.
4. At least 1 data generator. Generator rates should be configurable.
5. Generators of request workload should be implemented. The storage should be able to insert and process select queries simultaneously.
6. Monitoring system.
7. Test scenarios should be implemented as a demo of the solution.
8. **Bare-metal deployment is forbidden. Deploy the storage with replicas, queues, generators, requesters and other modules using Kubernetes, docker (see: Kubernetes' StatefulSet / Deployment / Job, etc.)**

Extra points:

1. Spark integration. You have to be able to read/write data from/to the warehouse using Apache Spark.
2. Comparison analysis. You have to compare possible multi-model warehouses and choose the most suitable.
3. Warehouse performance tuning.
4. Kafka Streams as a processor of incoming records.

Deadlines (for all projects):

0. **25.10**: Assemble the team and assign the roles.
1. **08.11**: Present the intermediate result (up to 15 scores*):
   - overall system design, chosen approaches and mechanisms
   - implemented data generators
   - deployment scripts and configuration
   - report as a presentation (both for course staff and other students)
2. **29.11**: Present the intermediate result (up to 30 scores*):
   - partial implementation of the solution (tables, views, etc.)
   - implemented ingestion process
   - implemented requesters, performance measurement and results verification tests
   - report as a presentation (both for course staff and other students)
3. **20.12**: Present the final solution (up to 35 scores*):

- full solution (including sharding and replication)
- full test scenarios
- **final report AND final presentation (both for course staff and other students)**

* - approximate preliminary scores. Presentation rules of thumb will be published later in the discord server. Maximum score is 100. 20 - 30 scores reserved for extra bonuses. Conditions for extra bonuses will appear soon. No presentation on the deadline day (except for the final solution deadline) requires presentation on the next deadline day with minus 30% of score (as a fine).

30 - 50 scores: 3 (both for exam and course project)
50 - 80 scores: 4 (both for exam and course project)
 > 80 scores: 5 (both for exam and course project)