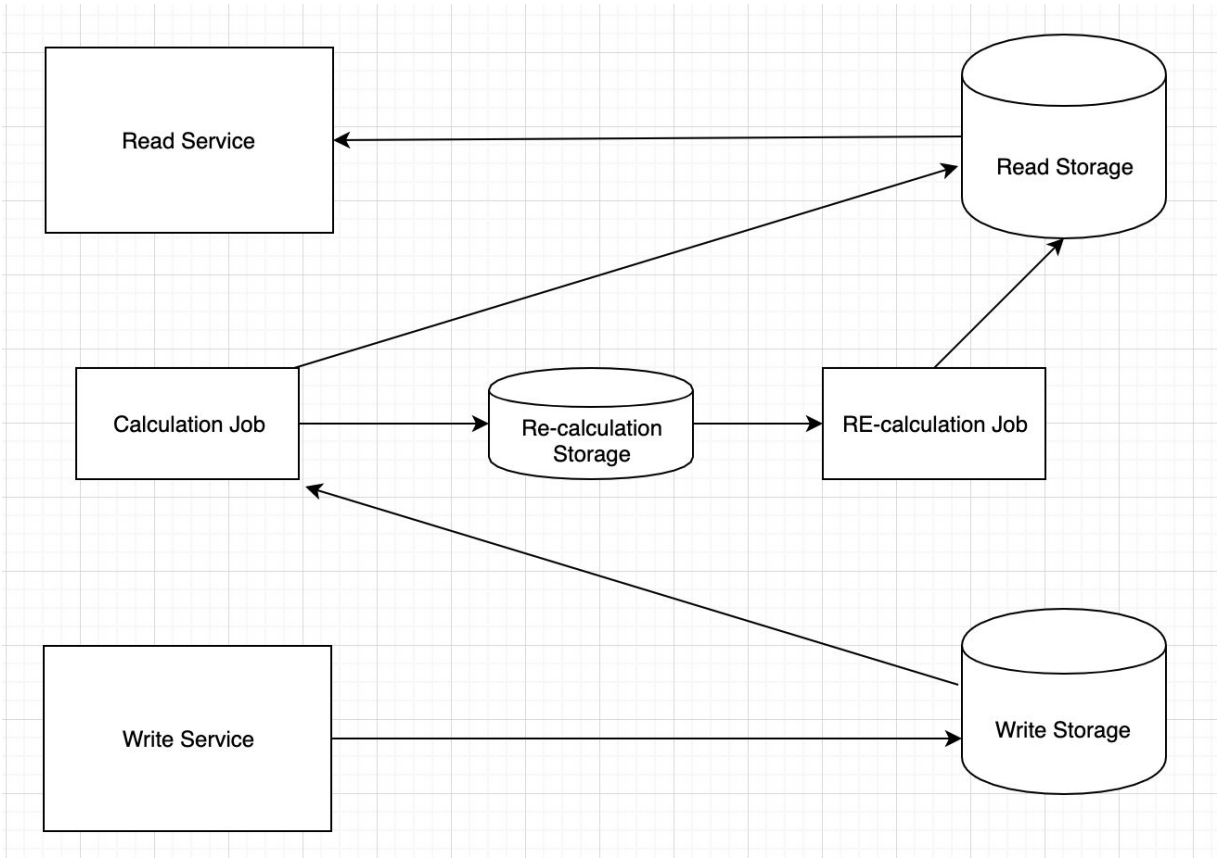
1. Overview
   The top-level design is like below:



   It does not show specific implementations. It shows the main parts of the architecture and the system's dataflow.

   The details are below.

2. Write Service
   a. Functionality
      The write service is a web api, which takes writing queries from the web/applications. It writes the raw data to the Write Storage.

   b. Extensibility
      It is important to keep the write service stateless, so that we can easily scale it out.

   c. Implementation

All Web-based frameworks can be considerable, such as SpringBoot, Ratpack, and JerseyMVC. However, I personally recommend the SpringBoot as framework and Ratpack as web-container.

When Scaling out is under consideration, a load balance mechanism (such as F5 or Nginx) should exist, or we can consider a clustered structure with Kubernetes for deployment.

3. Write Storage
    a. Functionality
    The write storage keeps the raw data. It does not contain complicated business logic. When a complicated calculation/analysis is required, we can access the raw data from it and put the result to somewhere else (i.e., the Read Storage in the later section)

    b. Extensibility
    It depends on the implementation. If we choose Oracle DB, a RAC can be considerable. If NoSQL solution is the case, then we can take advantage of its built-in scaling mechanism, for example, Zookeeper for HBase.

    c. Implementation
    Although both NoSQL and SQL is considerable, I will recommend an RDBMS solution, such as MySQL or Oracle DB. The relational database helps us easily access data in whatever way we like.

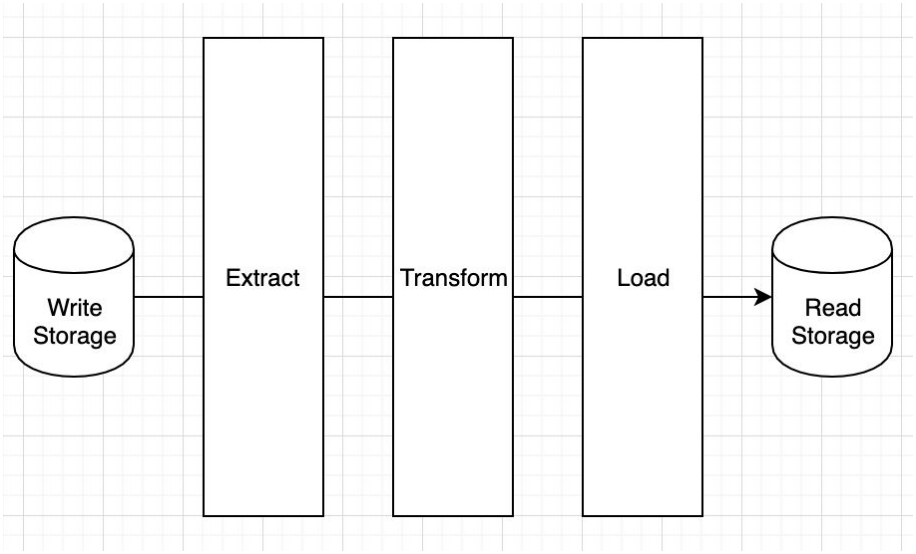
4. Calculation Job
    a. Functionality
    The calculation job is a scheduled job that:
        - Read raw data from Write DB
        - Do analysis
        - Store results to Read DB

    It does not have a web interface, unless we plan to control it's behavior during runtime.

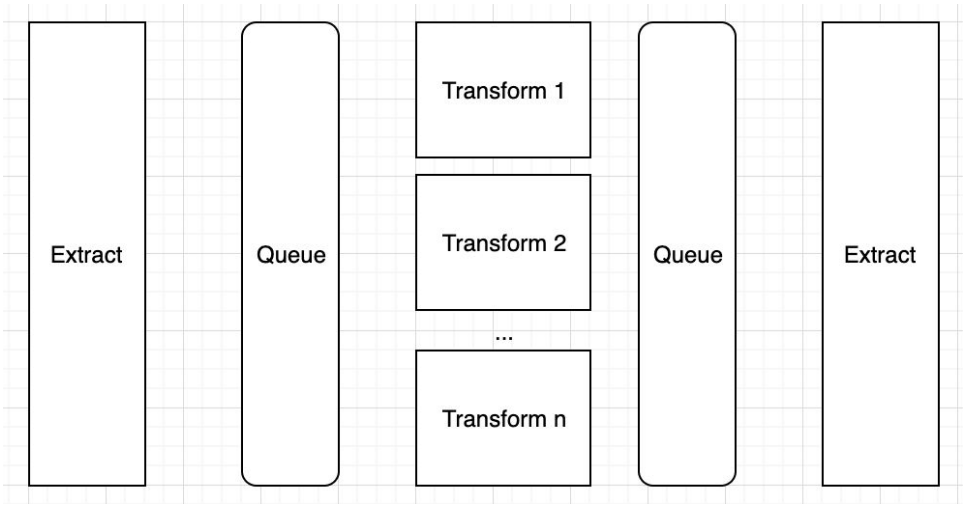
    b. Extensibility
    If extensibility is necessary, we have to make sure the job works based on a "partitioned" design. It should take care of which partition it is working for, otherwise the data can be redundant or deficient.

c. Implementation

An ETL (Extract-Transform-Load) structure can work. The “E” part reads from the write storage, the “T” part does the analysis, and then the “L” part put the result to the read storage.



Depending on the context, here the three parts can be in one application, or three different procedures. The three-procedure solution is considerable when the loading of the three parts are very different (e.g., when the “T” is very hard loading) or the business logic is very complicated (e.g., when we have 30+ analytical jobs to do)



If a three-procedure solution is decided, a remote queue service is needed between them, i.e., Kafka or RabbitMQ.

5. Read Storage
    a. Functionality
    It stores the analyzed data. When the front end needs, the backend service accesses this storage and returns to the front end.

    b. Extensibility
    Similar to the write storage, the read storage's extensibility depends on its implementation.

    c. Implementation
    Although both NoSQL and SQL can store data, here a NoSQL implementation such as MongoDB or HBase is more preferable. The business rules can change anytime, so the key-valued storing schema is suitable.

6. Read Service
    a. Functionality
    It takes queries from the front end service, access data from the read storage, and then returns back to the front end.

    b. Extensibility
    Like the read service, keeping the read service stateless is important to its extensibility.

    c. Implementation
    Any web-based framework can be considerable.

7. Re-calculation Job and Re-calculation Storage.
    When errors happen, we can keep the original data in an independent storage, and schedule another job to re-calculate it later.

    Regarding re-calculation, actually we have to options: we can either have the re-calculation job and storage independent from the ordinary calculation, or make them independent. It depends on the frequence and system loading.

8. Conclusion

The design is not fixed. Different decisions can be made due to different loading, complexity of calculation, budgets, etc. For example, if later we find that some queries repeats, we can have a remote-cache service, such as Redis behinds the read service, so that it doesn't have to always ask the same data from the storage.

It is important to keep these components low-coupling, so that changing design is possible afterwards. The low-coupling design is also crucial to short downtime (or, ideally, no downtime).