18. Job Scheduling System

1. Introduction

When a limit-sell order or some code is scheduled to run in the future, you need a piece of software like Apache Airflow that runs tasks when certain conditions are met. Because of its wide popularity in the industry, job scheduler has been among the top topics of system design interviews. In this article, I want to share with you my design of large-scale job schedulers based on my own experience and discussions with other engineers.

1.1 Requirements

In an interview, it is critical to lay down concretely the desired features of the system. No one can design a full-fledged system in an hour, so narrow the topic as much as possible! For the simplicity of the discussion, I'm going to focus on the core features of job schedulers:

- Create/delete a new job with its schedule
- Query all jobs owned by the user
- Query the status of a job (running, failed, finished, etc)
- Query the execution history of a job
- Retry support for failed tasks
- On-time execution (when a job is scheduled to run at 1 PM, it should be triggered around 1 PM)
- Needless to say, the system should be horizontally scalable and highly available.

1.2 Traffic Estimation

The traffic that hits a job scheduler varies in different applications. For internal-use schedulers, the traffic is small as the number of users is limited by the team size. On the other hand, client-facing schedulers can be bombarded with tons of traffic (such as schedulers in AWS). In this article, we assume the traffic is large for better generalization:

- We expect millions of active users.
- On average, each active user owns dozens of recurring jobs.
- On average, an active user creates a couple of new jobs per week.
- A large percentage of jobs repeat every day for a few times. Some jobs are executed with longer intervals.

Based on the estimations above, we can make the following conclusions:

- The backend database should be horizontally scalable, as the execution history grows quickly (~ 100 million rows of data created every day).
- The system is read-heavy, as a job is created once and read many times by the scheduler. The same holds true for execution history, an entry is created and can be queried many times by the user.
- We need a distributed group of workers to run jobs concurrently with varying capabilities.
- All critical services should be replicated to handle the large traffic.

2. High-Level Design

2.1 Database Design

In database design, the access pattern of your application determines what schema is used. Let's investigate the requests that hit the database:

Read Operations

- Given a user ID, retrieve all jobs that belong to it (by client)
- Given a job ID, retrieve all/latest execution histories belonging to it (by client)
- Find all jobs that are scheduled to run right now (by internal servers)

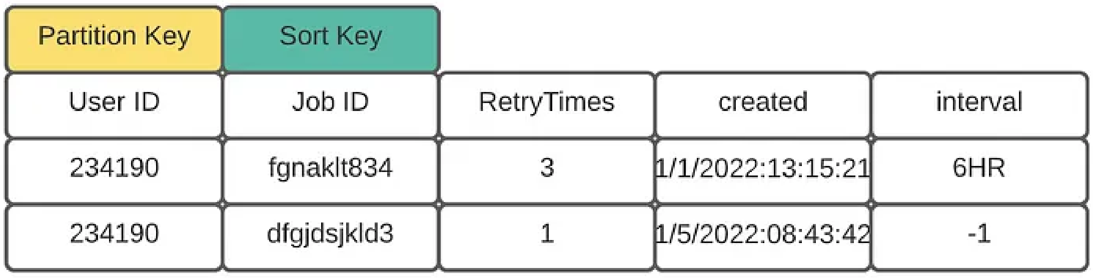
Write Operations

- A user can create/delete a new job schedule (by client).
- The workers will add execution histories to the database (by internal servers).
- The system updates the next execution timestamp of a job after running it (by internal servers).

In our use case, complex relational queries are not used. Most data access can be described as primary-key queries (e.g. given a job ID, get all executions). Also, the issues of strong consistency and transactions are not of paramount importance. Hence, both sharded SQL and NoSQL databases can handle all the requirements of the system as long as they are tuned for reads. For simplicity, I'm going to use NoSQL (Cassandra) language in the schema discussions, but please remember that you can choose other databases like Postgres or HBase.

Schema

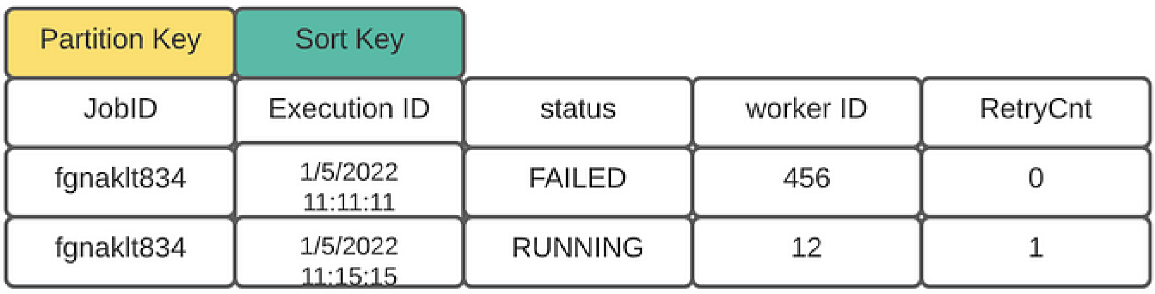
Job Table: We need a table that keeps track of job metadata such as owners, execution intervals, and retry policies. Remember the access pattern for this table: given a user ID, get some job records. Hence, we can use UserID as the partition key, and JobID as the sort key



Partition Key	Sort Key			
User ID	Job ID	RetryTimes	created	interval
234190	fgnaklt834	3	1/1/2022:13:15:21	6HR
234190	dfgjdsjkld3	1	1/5/2022:08:43:42	-1

Job Table

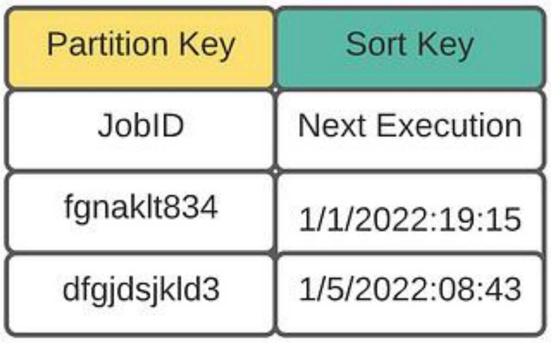
History Table: This table is used to store execution details of a job. Given a job, there could be multiple executions associated with it. Remember the access pattern: given a JobID, retrieve all execution histories. Hence, we use JobID as the partition key and execution ID as the sort key (the execution ID can be a simple timestamp).



Partition Key	Sort Key			
JobID	Execution ID	status	worker ID	RetryCnt
fgnaklt834	1/5/2022 11:11:11	FAILED	456	0
fgnaklt834	1/5/2022 11:15:15	RUNNING	12	1

History Table

Schedule Table: The core feature of any job scheduler is, of course, running jobs on time. Therefore, we need a data model that makes it easy to filter jobs by execution time. Below is a simple design that works well for a small amount of data:



Partition Key	Sort Key
JobID	Next Execution
fgnaklt834	1/1/2022:19:15
dfgjdsjkld3	1/5/2022:08:43

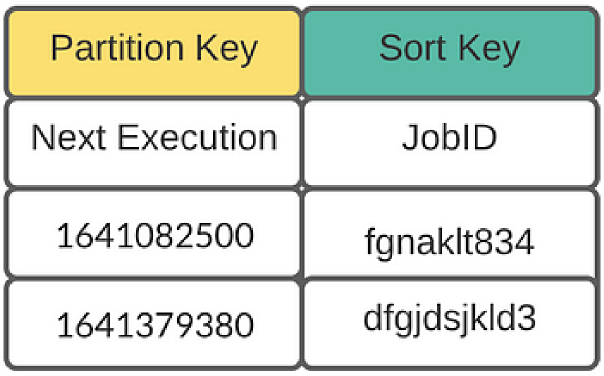
Schedule Table Design

To get the list of to-dos, we can simply run a query like this every minute:

```
SELECT * FROM ScheduleTable WHERE NextExecution == "1/5/2022:08:43"
```

One huge issue with the current design is that each query triggers a table scan. If the data is large, additional mechanisms are needed to ensure the scalability of the system. See more in the Details section.

If you have experience with large-scale databases, the word table scan should alarm you, not to mention doing it every minute. Is there a way to filter jobs without checking every record? Yes! If we simply reverse the data model in above figure, we get:



Partition Key	Sort Key
Next Execution	JobID
1641082500	fgnaklt834
1641379380	dfgjdsjkld3

Schedule Table Design

Note that the execution time is converted to UNIX timestamp with minute-level granularity. This means that jobs that are scheduled to run in the same minute will share the same partition key. To get jobs that are scheduled to run right now, simply find the right partition with the current UNIX timestamp:

```
SELECT * FROM ScheduleTable WHERE NextExecution > "1641082500" AND NextExecution < "1641082580"
```

2.2 API Design

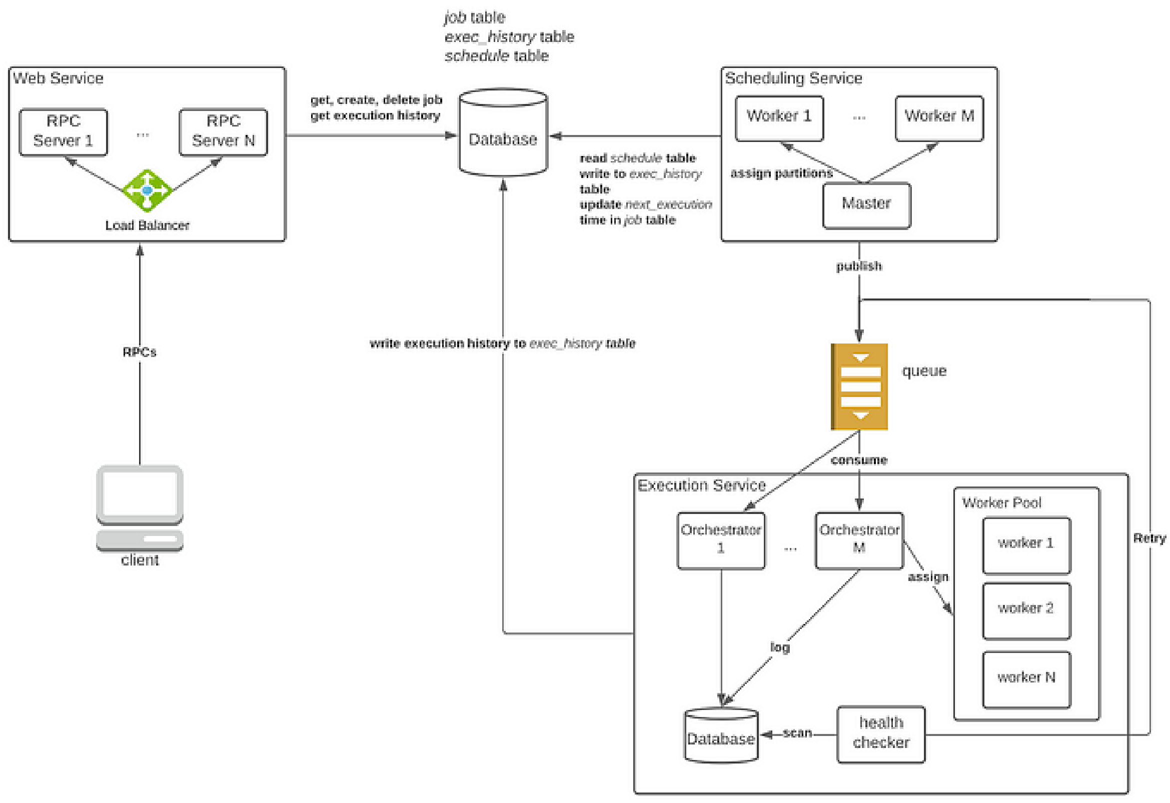
Our application is easy to use, it only needs the following RPC interfaces:

```
submit_job(user_id, schedule, code_location)

retrieve_all_jobs(user_id)

delete_job(user_id, job_id)

get_exec_history(job_id)
```

2.3 Architecture

By now we’ve laid down a solid foundation for the application the database schema, the RPC calls. Now is prime time to do some service-oriented designs!

- **Web Service**: The gateway to the scheduling system. All RPC calls from the client are handled by one of the RPC servers in this service.
- **Scheduling Service**: It checks the database every minute for pending jobs and pushes them to a queue for execution. Once a job is scheduled, create an entry in the execution history table with status = SCHEDULED. With this service, we guarantee that all jobs are pushed to the queue in a timely manner.
- **Execution Service**: In this service, we manage a large group of execution workers. Each worker is a consumer and executes whatever jobs it gets from the queue. Additional bookkeeping is needed to ensure re-execution upon worker failures.



High-Level Architecture

3. Details

If you read figure 6 in detail, you might be wondering about the purpose of the master in the Scheduling Service or the health checker in the Execution Service. In this section, we are going to discuss the necessity of these components and why do they work.

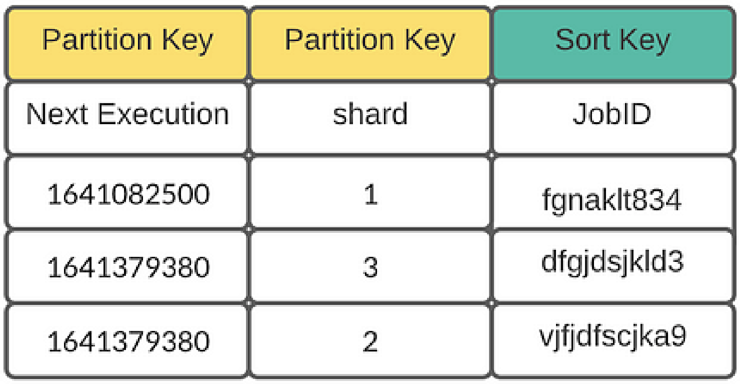
3.1 Run tasks on schedule

Let’s examine the query statement again:

```
SELECT * FROM ScheduleTable WHERE NextExecution > "1641082500" AND NextExecution
< "1641082580"
```

When the number of jobs running concurrently is small, the query yields a reasonable amount of data. However, if, let’s say, 100K jobs are scheduled to run in this minute, we certainly need more workers to handle the ingress data from the query as well as push messages to the queue.

The complexity arises when multiple workers are introduced — How can we distribute the data so that each worker only consumes a small fraction (10%) of the 100K jobs? It turns out that a simple composite partition key can fix this issue:



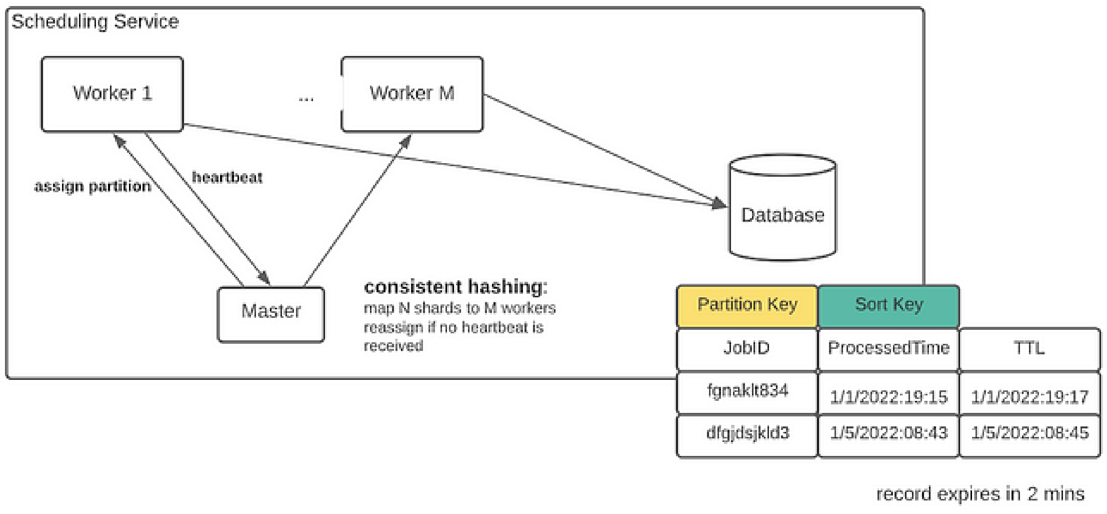
Partition Key	Partition Key	Sort Key
Next Execution	shard	JobID
1641082500	1	fgnaklt834
1641379380	3	dfgjdsjkld3
1641379380	2	vjfjdfscjka9

Schedule Table with Composite Partition Key

When a row is added to the schedule table, it is randomly assigned a shard number. With the new schema, it is super easy to distribute the load across workers:

```
Worker 1:

SELECT * FROM ScheduleTable WHERE NextExecution > "1641082500" AND NextExecution <
"1641082580" AND shard =1

Worker 2:

SELECT * FROM ScheduleTable WHERE NextExecution > "1641082500" AND NextExecution <
"1641082580" AND shard = 2
```

Although the additional partition key makes it easy to shard data, there is another wrinkle — workers come and go all the time. There is no guarantee that all shards will be covered. The problem becomes how can we assign N shards evenly to M workers, where M can change at any time?



Partition Key	Sort Key	
JobID	ProcessedTime	TTL
fgnaklt834	1/1/2022:19:15	1/1/2022:19:17
dfgjdsjkld3	1/5/2022:08:43	1/5/2022:08:45

Scheduling Service in Detail

To make sure a complete work assignment, we can borrow some ideas from MapReduce, where a master is used to assign and monitor workers. If a worker dies, the master will resend its work to some other nodes. An additional local database is used so that no job is scheduled twice. When a job is pushed to the queue, an entry is created in the local DB with 2 minutes of expiration time. If the original handler of the record dies and the shard is handed over to another worker, the new worker will skip tasks that exist in the database.

3.2 Queue delivery

A queue is introduced for two reasons. Firstly, we want a buffer that holds all pending jobs at peak hours. Secondly, it decouples the two services. The execution service can increase/decrease its capacity without coordination with the scheduling service. However, with the additional middleware, we must consider what type of delivery works best for the system (Assume the queue is implemented in Kafka).

At-least-once delivery

In this mode of delivery, messages are processed before their indices are committed by the consumer. If the consumer dies before committing, the same message will be processed again by a different worker.

At-most-once delivery

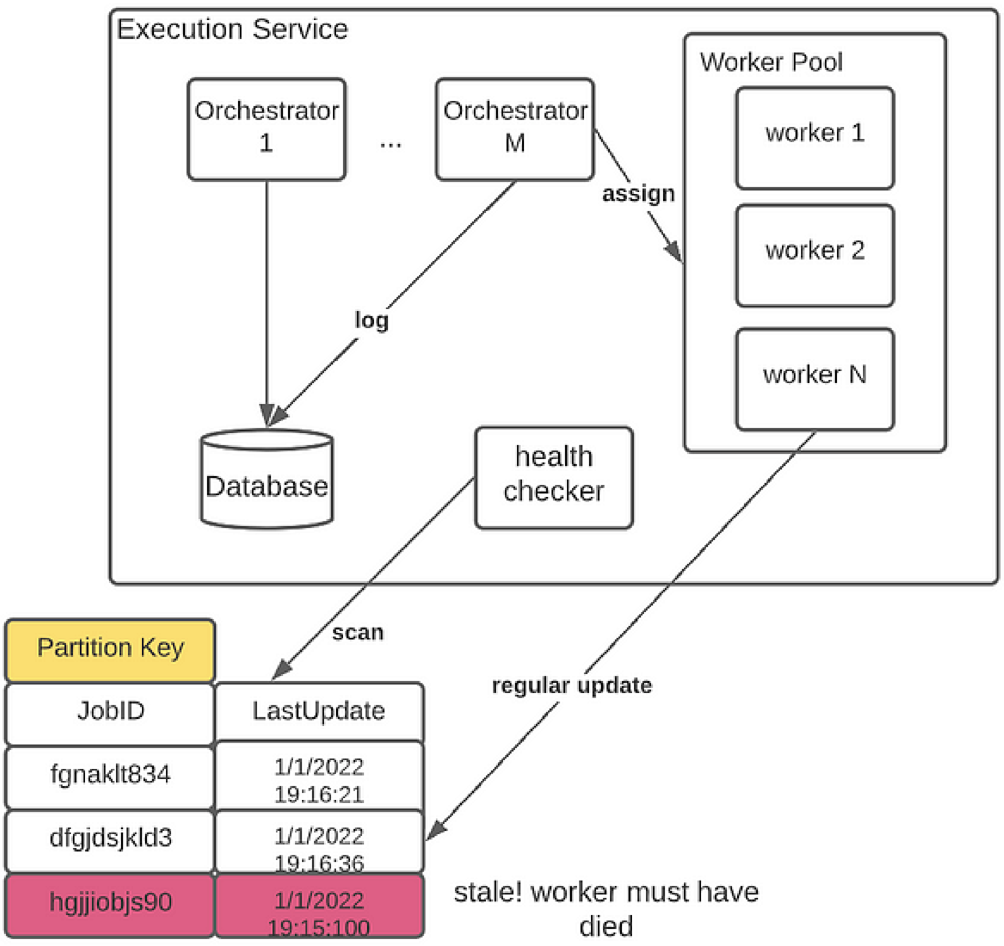
In this mode of delivery, messages are processed after their indices are committed by the consumer. If the consumer dies before finishing the task, the job won’t be reprocessed.

In our application, at-least-once delivery is more fault-tolerant and makes it easy to retry failed tasks upon orchestrator failure. A message (job) is committed only when the orchestrator successfully starts a container on one of the workers.

3.3 Handling failures & Retries

Although at-least-once delivery guarantees that every job gets assigned to a worker, it does not prevent jobs from getting dropped upon worker failure. To achieve true retry capability, an asynchronous health checker is introduced to run the following logic:

1. When a job is assigned, an entry is created in the local database with the latest update time.
2. All workers are required to refresh the update time ~10s.
3. The health checker will periodically scan the table, looking for jobs with stale update time (e.g. older than 30 seconds).
4. Jobs that meet the above criteria are sent back to the queue for re-execution.



JobID	LastUpdate
fgnaklt834	1/1/2022 19:16:21
dfgjdsjkld3	1/1/2022 19:16:36
hgjjiobjs90	1/1/2022 19:15:100

Failure Handling

3.3 Data flow

With all the wrinkles in the high-level design address, we can finally come up with the data flow of the system:

Create/delete job/Retrieve history

- The client sends out an RPC call to Web Service.

- One of the RPC servers queries the database using the provided partition key and return the result

Schedule a job

✓ Master

- Every minute, the master node creates an authoritative UNIX timestamp and assigns a shard ID (see details for more) to each worker along with the timestamp
- Check worker health regularly. If it dies, reassign its work to others

✓ Worker

- The worker queries the database with the timestamp and shard ID.
- For each row, send it to the queue if it has not been scheduled (see details for more)

Execute a job

✓ Orchestrator

- A group of orchestrators consumes messages from the queue
- Given a message, find one worker with the least workload. Assign the job to the worker
- Commit the index, repeat steps 1 to 3

✓ Worker

- The worker regularly update the local database with its timestamp

✓ Health Checker

- Scans the local database ~ 10 seconds
- If any row hasn’t been updated in ~30 seconds, retry it by pushing the job ID to the queue