Architecture of a Massively Scalable Distributed ETL System

Gautam Karnataki
Contents

Abstract 3
Limitations of Scaling-Up 4
Enter: Distributed ETL Approach (Scaling-Out) 6
Other Challenges 9
Conclusion 10
Abstract

The Need for a Distributed ETL system

An Extract, Transform and Load (ETL) tool needs to be robust, scalable, high throughput and fault tolerant. Very much like an e-Commerce transaction system. Designing such a system on a distributed computing backbone can be extremely rewarding, given that mid-size to large organizations might be collecting data from multiple sources and bringing it all together into an integrated warehouse—resulting in thousands of batch and real-time jobs running during the course of a day.

For example, retailers collect inventory, sales, finance, marketing, clickstream, and competitor data multiple times a day. But aggregating this data by running ETL jobs, only once daily, can slow down decision-support systems and rules engines, which must feed essential decisions (like dynamic prices) back to the system to control demand.

For many e-commerce analytics and data-mining solutions, a slow ETL tool might prove to be a huge bottleneck. While commercial and open source tools help implement such workflows, it is often better to consider a homegrown ETL tool based on good design and distributed-computing principles.

Let us find out how to build your homegrown ETL solution and use a task queue to scale the tool horizontally.
Limitations of Scaling-Up

At its core, an ETL job can be described using a DSL or just a plain XML/JSON structure. For example, the following is a simple ETL job that copies data from Amazon S3 to MySQL:

```xml
<job>
  <source-type>S3</source-type>
  <target-type>MySQL</target-type>
  <from>
    <location/>
    <aws-credentials/>
  </from>
  <process/>
  <to>
    <database/>
    <host/>
    <port/>
    <query/>
  </to>
</job>
```

An example of this is the predefined language structure of hamake - https://code.google.com/p/hamake/

Accordingly, it is possible to code up a parser and an ‘interpreter’ that run the job from an ‘ETL Box’. A programming language like Python or Java can be used to spawn multiple threads and utilize all the resources of the box. That way, hundreds of jobs can be added and processed by the ETL box.
A single-machine ETL Box design (see fig. 1) will work until the box either runs out of memory or disk space, or potentially becomes too slow, as a single CPU is 100% utilized leading to job starvation. This architecture sets serious concurrency limitations—if (desired concurrency > number of cores), then the CPU starts time-slicing between jobs, leading to performance issues. Such problems can be resolved by scaling up to a system with better hardware—more cores, more RAM, and bigger hard drives.

However, as organizations grow and the need for more data at faster rates increases, the above approach will not always work. Jobs running on a single machine will always compete for resources, compromising on reliability and fault tolerance.
Enter: Distributed ETL Approach (Scaling-Out)

Distributed computing is all about scaling—effortlessly. A well designed distributed computing system will mostly have the following core characteristics in addition to others.

a. Addition/removal of resources as needed – CPU, memory, storage
b. Massive parallelism
c. Fault-tolerance and reliability

An ETL system can be modeled on a distributed architecture where a task is analogous to an ETL job. Job execution can be controlled by a scheduler—whose primary responsibility is to ‘push’ ETL jobs to worker nodes at pre-defined intervals. A suitable queuing structure can be used as transport.
For Python-based ETL tools, Celery is a great choice for facilitating distributed task execution. Celery comes bundled with its own scheduler (Celery Beat), a highly useful job-monitoring system (Celery Flower), connectors for multiple transports (among them, RabbitMQ, Redis are the most popular), and a hoard of configuration options for optimizing workflows—including setting concurrency limits, automatic job retries, pre-fetch counts, non-blocking IO and many others.

Now it’s time to optimize.
In data warehousing, the following scenarios are possible:

a. Some jobs need to run more frequently than others
b. Some jobs fetch, process, and load more data than others
c. Some jobs are process intensive (complex transformations), some are IO intensive, and some are both, process and IO intensive
d. Some jobs need to be processed on priority, although they enter the queue much later than other jobs

These scenarios amplify the need to treat different jobs differently.
A queue is a FIFO. Workers fetch jobs from the queue, one at a time, and process them (unless you have PREFETCH option > 1). The above cases demand that each type of job be pushed into a different queue, meant only for a specific purpose.

Queue A is a dedicated queue for all priority-1 jobs. Those jobs are short and need to be executed on at much shorter periods than other jobs. Further, Queue-A jobs are channeled only to worker W1.

This structure is repeated for each worker in the mix, and multiple queues are created to set up a Queue->Worker relationship.
Creating a distributed setup (fig. 4) has the risk of opening the destination database to too many connections. This can lead to a clogged target database and suboptimal performance. This challenge can be solved by introducing distributed semaphores. A distributed semaphore works in the following way:

a. A semaphore queue is filled up with as many dummy messages as the number of connections allowed on the target resource.

b. When a worker wants to write to the target resource, it checks the semaphore queue and acquires a message.

c. If there are no messages in the queue, the worker believes that the maximum number of concurrent connections on the resource is already reached. It makes the worker wait for another worker to free up a connection.

d. If the worker is successful in getting a message from the queue, it executes the job and releases the message back to the queue post execution of the job.

e. If a worker crashes while executing a job, it throws an exception. The exception-handling mechanism takes responsibility to release the message back to the queue.
Implicit Dependency Resolution

Sometimes a job gets stuck in a worker due to an incomplete, dependent job. This holds up the worker unnecessarily. In the worst case, all workers can get held up, with each worker holding up a job which cannot be executed—because it is waiting for some other job to finish.

In a strict sense, it is not a deadlock, but a kind of 'jam' (as in traffic jam).

A simple solution is to pre-empt the job from the worker and send the job back into the Celery queue. This is done by raising an exception whenever the worker senses that a job is waiting. It then raises an exception.

Conclusion

Re-architecting a system so that it gets the features of a horizontally scalable distributable system, is all about creating a design similar to the one above.

The system needs to have the ability to store data (jobs) and compute (worker execution code). Much like a map-reduce job, every worker needs to have copies of the same code, as well as the job JSONs.

A master node picks the name of a JSON and passes it to the queue. The JSON name is therefore a message in the queue. A Celery round robin algorithm decides which worker will pull the message. The worker that pulls the message only has to execute the custom code for the job using something like ‘execute (json-name)’.

This architecture can scale to any number of nodes. More stability can be built in using queue-mirroring techniques that ensure a queues state is maintained even in case of failures. The architecture can then be deployed on either ‘on premise’ systems or cloud-based (EC2) systems.
About Tavant Technologies

Headquartered in Santa Clara, California, Tavant Technologies is a specialized software solutions & services provider that delivers impactful results to its customers across North America, Europe, and Asia-Pacific. Founded in 2000, the company employs over 1500 people and is a recognized top employer.

Find Tavant Technologies on LinkedIn and on Twitter.

About the Author

Gautam Karnataki is a Big Data and Enterprise architect at Tavant Technologies. He comes with 17+ years of experience in developing scalable architectures and platforms, working with e-Commerce, Digital Marketing, Consumer Lending and Manufacturing domains on various technologies.

Contact Information

Tavant Technologies, Inc.
3965 Freedom Circle, Suite 750
Santa Clara, CA 95054
United States

/TavantTechnologies /Tavant /Tavant-technologies /TavantTechnologies