Dron - an integration job scheduler

Dron - an integration job scheduler
- 1. Use cases
- 2. Problem Definition
- 3. Thoughts That Have Occurred to me During the First Iteration
  - 3.1. Scalability Requirements
  - 3.2. Design Overview
  - 3.3. Erlang Issues
  - 3.4. Design Questions
- 4. Dependencies Language
- 4. Failures
- 5. Job Definition
- 6. Publisher/Subscriber
  - 6.1. RabbitMQ
- 7. Evaluation
  - 7.1. Tests to be conducted
  - 7.2. Other systems

1. Use cases

1. Facebook uses an entire suite of systems in order to conduct data analysis. For example, Scribe is used to move data from their web tiers to HDFS. Moreover, jobs that scrape the federated MySQL databases containing all website related date are run on daily basis. Lastly, data is transformed into Hive readable format and log analysis is conduct. However, since the amount of data copied varies from hour to hour, data analysts do not have an easy way of conducting their experiments. Moreover, we do not want an analyst to create a workflow with a step that scrapes the data.
2. Build a recommendation system. Have a job that is triggered when new products are added to the system (HDFS). Then a Mahout job is triggered, that reads the file and updates the MySQL tables containing item similarities.
   http://ssc.io/deploying-a-massively-scalable-recommender-system-with-apache-mahout/
3. One example to capture multiple frameworks (e.g. HDFS importer, Dremel, Hive)
4. One example to capture cache usage

2. Problem Definition

1. What is the nature of most of the jobs?
   The majority of jobs will be data intensive. They will either move data from one storage to another or they will run queries on data.
2. Will the profile of jobs be batch or stream?
   The current state of the art data analysis is currently conducted with batch jobs.

3. Is all the data created with Dron jobs?
   No. External programs/frameworks can create HDFS tables. This is the more general case. Since the system will work in the open-world, I will have to figure out a way of filtering notification events.
4. Should the system have a cache mechanism?
   This sounds to be a good idea. Frameworks will be able to save their output in case, and then subsequent jobs will be able to read it from there.
5. Is a framework supposed to know how to read the output of another framework?
   I think that this depends on the frameworks and that initially the system should not handle it. An option would be to provide libraries that can be implemented for each framework mapping. However, most of the frameworks should be able to cooperate (e.g. Hadoop, Hive, Mahout)

3. Thoughts That Have Occurred to me During the First Iteration

3.1. Scalability Requirements

First of all, I had to define the goals of the system in terms of scalability. It should be able to support the following:

- scale up millions of job
- as it scales up it should not introduce delays in scheduling
- it should not have any single point of failures
- should not be bound by the scaling capacity of a DB (e.g. Mnesia can only store tables of up to 2GB)
- it should not restrict the implementation of dependencies (i.e. think about the problems that may be raised by a partitioned dependency graph)
- it should not restrict the possibilities of optimizing job scheduling using the dependencies DAG

3.2. Design Overview

While brainstorming for various designs I went over several approaches:

1. A simple master that has his state saved in the database and that manages a set of workers. The problem with this approach is that the master can not scale up to millions of jobs. Moreover, it can only support a cold stand-by, which basically means that the state of the server needs to be reconstructed from the DB in case of a failure.
2. Add a Zookeeper quorum to the system so that in case of failure a hot standby server can continue scheduling. However, this approach still does not solve the scalability issues imposed by having a single DB.
3. Have a shadowed master that uses a fragmented(sharded) database. This has the advantage of scaling up by just adding nodes to the database pool. However, while scheduling a job we always have to use the network (even for simple reads). This approach may impact the scheduling performance of the system. As a solution to this approach we may try to reduce our number of read/writes to the database. (NOTE: I should also check how Mnesia handles the distribution of load. I am not sure if they use consistent hashing).
4. Create a very thin master that only handles load distribution using consistent hashing. Subsequently, we have another layer of coordinator machines that each have a

database. A coordinator machine is supposed to schedule 1 million jobs. When a coordinator fails, the jobs must be scheduled by one or more other coordinators. We can just move the jobs to one of the database replicas. However, this approach may create hot-coordinators. Another approach is to distribute the database across the remaining set of coordinators.

In order to provide a hot-standby we have four options:

1. use Zookeeper
2. for every operation pass messages to the shadow masters
3. use a database to synchronize the system
4. implement own Paxos style synchronization and leader election

3.3. Erlang Issues

Add a paragraph explaining the decisions I have made while exploring the supervisor tree. The fact that the supervisor was not designed to work on a distributed environment seriously impacts the design. I will have to manually implement a heartbeat mechanism. Supervision will only be implemented locally on each worker node.
How am I going to handle network partitions? Think about the case in which I am not receiving a heartbeat from a worker because of a partitions. However, the worker may be still up and running, handling all its jobs. In this case I will end up with two instances of the same job. The pragmatic and easy solution is to have a big heartbeat expiration time. (e.g. 10 minutes)

http://www.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/ (Further reading may be interesting)

In the end it seems that I do not have to implement the heartbeat protocol manually as Erlang has tick time for monitor_node. Talk about it.

Erlang does not have a system to create a unique id. I have experienced with dirty_update_counter which only works on one node. However, I have decided to use a general id generated using {node(), erlang:now()}.

Erlang transactions can deadlock. Thus, the retries argument is used to retry the transaction. Its default value is infinity.

Explain why I have found gen_fsm and gen_event not useful. Moreover, explain why I had to redesign the way I was making use of gen_server (It is very slow in passing big states because it keeps on copying them).

Talk about the transition from passing state around in gen_server to using ets.

Talk about the recovery of dron_pubsub and dron_pool (I just read from the database the workers). However, I will have to implement something to figure out when a node is down.

Talk about fault-tolerance achieved using gen_sever (refer to the paper). Does it support

network partitions? It does not seem to be able to do it.

Gen_leader - is crooked because in my opinion it is not implemented properly. For example, the broadcast is not done atomically. Thus, if the master fails, the slaves can be in different states. It would help to know how the Erlang run-time works (how it stores the messages in inboxs).

Mnesia running on a single node is overloaded when 1k JI/sec are running. This is because an Mnesia node can not run more than 1400 transactions at a time. Whenever a transaction is run and ets table is created and deleted. However, in Erlang there is an option (ERL_MAX_ETS_TABLES) that bounds it. Theoretically this option could be increased but I am not sure how it affects the dump of transaction log.

Some information about Dron1:

- 8 transactions are conducted for each job instance
- a job instance(no deps) occupies 154 w => 1.5 million job instances occupy 1.72GB
- a job(no deps) occupies 351 w
- because of this the system only supports 500 ji/s
- I have changed get_job and get_job_instances to do dirty_reads. However, this only reduces the amount of transactions by 1 (to 7).
- Calls (2 store_job_instance, 1 set_job_instance, 2 store_worker, 1 get_dependants, 1 set_resource_state) - how many does it support now?
- increase erl_max_ets_tables, erl_max_ports (when many workers), +P number of processes

100	ERL_MAX_ETS_TABLE=1400, ERL_MAX_PORTS=1024, +P32768	Crashes with too many db tables open!!!
120	Same as above	Managed to reduce the number of transactions per job to 7 (instead of 8). I used dirty_read
300	ERL_MAX_ETS_TABLE=65536, ERL_MAX_PORTS=16384 +P256000	The master simply crashes. However I should run the test on amazon. I suspect long calls? Tough not sure. MUST ANALYZE!
400	when I removed the ets:lookup for max delay!!! It was returning badarg	
500	when I changed the workers to be in memory as well	It dies because get_worker call timeouts!!!! The reason is that there are too many messages for a single process to handle => 1) parallelize get_worker..not really possible because it would imply a DB. However, since the

		load is so high ..the db transactions will keep on retrying 2) optimize it..can't really see how..seems ok

What about Mnesia is overloaded?

Another interesting measure will be to check how the performance gets affected on mnesia with db in memory and with db on disk.

Coordinators have several issues

- we do not want to end up with coordinators scheduling jobs stored on different nodes
- use the same hash as in mnesia
- this does not solve the problem of dependencies!!!!
- change the hash in mnesia!!!
- I must carefully pick the hash as I may end up with hotspots for the core jobs that have many dependants.

Evaluate this by testing my own function against the default one!!!

Debate about the solution with coordinators and a single DB:

- I rely on Mnesia to access data. It may not scale if there are many jobs. I don't think there is a way to see it..without trying.

Problem with N schedulers handling the workers. I can not go for the solution in which every scheduler has access to all the workers because I will end up with a big bottleneck. No worker will be able to keep the data in-memory unless every single worker change will be broadcasted to all the schedulers (it's extremely wasteful and slow). Think about other options..... I've currently decided to go with the self-adapting separate worker pools.

Talk about various versions of worker failure handling..and scheduler failure handling.

Describe the old way of reconstructing the pool's state. The pool was read from the database. However, this stopped working once I have introduced multiple-schedulers. A solution would be to store together with the worker..the scheduler node that it's managing it. Another solution is for the coordinator to use it's in-memory state in order to determine them.

What are the ways of implementing the worker heartbeat?

- worker pools send their load every two minutes. Based on that the coordinator does balancing
- schedulers have policies based on which they decide the usage of the workers. Thus, they either send worker requests or offers. It is more flexbile, schedulers with different priorities can be added. Conduct experiments to find out the point where too many messages are passed around.

What is the role of the coordinator in the dependency module?

- it's the one that always knows which resource has been satisfied
- must notify the appropriate schedulers of the dependency that has been satisfied. There are two ways of doing it. Either let the coordinator to fetch all the dependants of a given resource and then notify the appropriate schedulers for each dependency. The second option is to send a broadcast to all the schedulers containing the RId of the satisfied resource. I think I prefer the second option as the number of dependants may be bigger than the number of schedulers. Another reason for not to go with the first option is because the master of a scheduler can change...thus we may end up sending notification to a dead node.

3.4. Design Questions

1. Who should write to the DB? (just the master node or the workers as well).
   Currently only the master node is writing so that we have better control of what is happening within the system. Moreover, we will avoid doing unnecessary reads to synchronize the state.
2. Should a worker keep a process open as long as a MapReduce job runs?
   No. I do not see any advantages for doing this. Moreover, the scheduler can be notified using the pub/sub mechanism and the functionality will already have to be in place.
3. Where should the job instance timeouts be handled?
   First, they were handled on the master node. However, after a while I have moved them to the workers. Depending on how I am going to handle worker failures the following scenarios can happen. The state of the worker is failed and we also assume that the job instances running on that machine have failed. Then having the timer on the master node does not bring any advantage. However, if I manage to keep the job instances running, then I will have to somehow reconstruct the timers.
4. How should I handle Mnesia transaction aborts?
   First, the causes of the abort must be identified. I think they can fail because of the maximum number of retries being achieved or because of an hardware failure (from Erlang doc: If something goes wrong inside the transaction as a result of a user error or a certain table not being available). I think no matter what I should create transactions retry.
5. How do I handle job instance timeouts for framework type of jobs?
   I should be able to use an API to kill a job instance. This is kind of crappy because it means that I have to couple Dron with the framework even more (e.g. framework should send a (jobName, dronJobInstanceId) and Dron should know how to kill it. For now, I will just assume that the instances are running as much as it takes to run in the framework.
6. What happens when a dependency can not be satisfied (e.g. a job instance on which it dependens has failed max_retries)?
   At the moment the job instance will stay in memory until its wait_timeout is reached.

4. Dependencies Language

1. **Based on job instance ids**
   TODAY, TODAY-N:TODAY+N, HOUR, MINUTE, SECOND, LAST_RUN-N:LAST_RUN+N

The problem with this part of the language is that it is data insensitive. It should probably be used mostly for expressing CPU intensive pipelines. It seems to be an uncommon use case.

2. **Based on data containers**
   Dron Memory - I should implement a distributed Dron memory API and holder. Kind off an overkill
   Local - can store the date locally on a specific worker. The problem with this option is that worker failures imply a computation restart. Moreover, we may end up with hot spots for popular jobs and the scheduling has to be adapted.
3. **Direct Streams**
   Job instances must be scheduled on the same machine.
4. Support queries: Who creates/alters a table? It should return the name of the job.

4. Failures

Two types of failures:

1. Erlang node VM failure
2. network failure

5. Job Definition

A new job type definition will involve:

1. module to transform notification events into job instance ids
2. module to handle job launch
3. change framework to send notifications

6. Publisher/Subscriber

6.1. RabbitMQ

1. How does it handle broker failures?
   If one uses a single broker then messages may be lost even if persistent queues are used. This happens because messages may be buffered before persistent (i.e. a small failure time-window). An option is to use publisher confirm (i.e. the client knows which messages have written to the disk). However, we still don't want to suffer downtimes or to loose messages in case of disk failure.
   One can use a cluster of RabbitMQ nodes with pairs of active/active nodes. For each mirrored queue there is one master node and several slaves. The slaves apply the operations that occur on the master in the exactly same order as the master. If a new slave enters the cluster then it only receives new messages and its state will become fully synchronized as consumers consume messages. Thus, if we have an unstable pool we have the possibility of not having any fully synchronized slave node. If master fails then the eldest slave is promoted. The effect of a network partition on a slave node is the same as removing it and adding it back to the pool. It will have to discard all its messages because it does not know whether its queue content has diverged from the master or not.
2. How to model notifications?
   Every framework should have a separate queue to send events. Subsequently, a

module that transforms/filter events into job instances will listen to the queue is handling. This solution does not require an event filter/dispatcher process. However, I will have to be careful since some queues may get very busy.

7. Evaluation

7.1. Tests to be conducted

1. Facebook test
   10000 job instances at a time and a rate of 30/second. This gets translated into 5000 long running jobs, 6000 minute jobs.

7.2. Other systems

DAGMan - DAGMan submits programs to Condor in a order represented by a DAG. The input is given as a DAG file. DAGMan is executed as a Condor job and it is responsible for making sure that the programs execute in the same order as the one specified in the DAG input file. Thus, it monitors log files to enforce the ordering. It is also responsible for scheduling, recovery and reporting.
A node in the DAG may contain more than than a job (i.e. it can be formed by a pre script, the condor job and a post script). The success of a node in the DAG is defined base on the pre and/or post script. Thus, in some cases if the user wants can hide the failure of the Condor job. (Here I should check what does mean to run a series of jobs in a Condor cluster). By having a pre and post script DAGMan moves the responsibility of defining a powerful language to describe dependencies to the users. Now, they can decide if the pipeline should fail or not in case of several job failures.

Oozie - contains two types of nodes:

- action nodes: map-reduce, pig, map-reduce streaming, file-system, java
- control nodes: decision, fork and join nodes

Work-flows are defined using a PDL in a XML file. This implies that the users do not have the possibility of defining dependencies out of their DAG.
I am not sure what they mean by “Oozie runs work-flow jobs under the assumption all necessary data to execute an action is readily available at the time the work-flow is about to execute the action”.
A problem is that the integration with other systems is not properly made. Whenever someone registers a MapReduce job he/she has to pass a link to the job so that when the job is done, the framework can use the link to notify. I think a more complex solution, such as frameworks notifying on events is a better idea.

What did I learn from the Oozie talk:
1. Hcatalog may provide a pub/sub system
2. Jobs should block when Hadoop down
3. Check Process Definition Languages
4. They provide two ways of notification: polling or event callback

Hadoop JobControl

- can programatically model dependencies between Hadoop jobs

- does not support retries but I think it should be difficult to add them
- no way to schedule the bundles more than once
- still can not create complex bundles