ETL pipeline to achieve reliability at scale

By Isabel López Andrade
### Accounting at Smarkets

<table>
<thead>
<tr>
<th>account_id</th>
<th>source</th>
<th>money</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>deposit</td>
<td>£20.11</td>
<td>2018-02-01 12:47:37.039161</td>
</tr>
<tr>
<td>1</td>
<td>order.execute</td>
<td>£20.11</td>
<td>2018-02-01 13:14:36.794810</td>
</tr>
<tr>
<td>3</td>
<td>order.create</td>
<td>£20.11</td>
<td>2018-02-01 13:14:36.794810</td>
</tr>
<tr>
<td>4</td>
<td>market.settle</td>
<td>£10.91</td>
<td>2018-02-01 13:25:08.737379</td>
</tr>
<tr>
<td>5</td>
<td>order.execute</td>
<td>£10.91</td>
<td>2018-02-01 13:28:20.156321</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>account_id</th>
<th>year</th>
<th>month</th>
<th>stake</th>
<th>deposit</th>
<th>withdraw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2018</td>
<td>1</td>
<td>£10.00</td>
<td>£50.00</td>
<td>£0.00</td>
</tr>
<tr>
<td>1</td>
<td>2018</td>
<td>2</td>
<td>£15.00</td>
<td>£10.00</td>
<td>£0.00</td>
</tr>
<tr>
<td>2</td>
<td>2018</td>
<td>3</td>
<td>£5.00</td>
<td>£0.00</td>
<td>£40.00</td>
</tr>
<tr>
<td>3</td>
<td>2018</td>
<td>4</td>
<td>£30.00</td>
<td>£0.00</td>
<td>£10.00</td>
</tr>
<tr>
<td>4</td>
<td>2018</td>
<td>5</td>
<td>£10.00</td>
<td>£0.00</td>
<td>£0.00</td>
</tr>
<tr>
<td>5</td>
<td>2018</td>
<td>6</td>
<td>£10.00</td>
<td>£10.00</td>
<td>£0.00</td>
</tr>
</tbody>
</table>
Accounting at Smarkets

Reports generated daily using transactions from the exchange.

In 2013, the average number of daily transactions was under 190K.

In 2018, this figure is over 8.8M.
Original pipeline

- Difficult to identify errors.
- Manual work to regenerate reports and expert knowledge of the system.
- System too slow and unable to scale. It took more than one day to run.
- Costly storage.
Requirements

- Fault tolerance and reliability.
- Fast io, availability, durability, and cost efficient.
- Good processing performance.
- Scalable.
Fault tolerance and reliability

Vulnerabilities

- Communication with exchange may fail.
- Hardware or software errors may happen while the job is running.

Design solutions

- Store transactions per day.
- Compute financial statistics per day.
- Retrieve the last two days worth of transactions.
- Break the accounting job into modular Luigi tasks.
class GenerateHumanReadableAccountingReport(AccountingTask):
    def requires(self) -> luigi.Task:
        return GenerateAccountingReport()
    def run(self) -> None:
        with self.input().operate('r') as target_path:
            df_accounting = pd.read_parquet(target_path)
        with self.output().open('w') as file_:
            df_accounting.to_csv(file_, sep='\t', index=False)
    def output(self) -> luigi.Target:
        return self.get_target(path='data/reports/accounting-report.tsv')
Efficient storage

- Columnar storage.
- Only read the columns needed for the task.
- Minimised I/O.
- Efficient compression and encoding.
- Python support.

Parquet
Efficient storage

- High durability.
- High availability.
- Low maintenance.
- Cost efficient.
- Decoupling of processing and storage.
- Python library boto/boto3.
- Web interface.
Good performance

Requirements

- Fast data processing.
- Scalable.

Solution

- General purpose data processing engine.
- Massive parallel. Spark builds its own execution plans.
- Caches data in RAM.
- Python support.
Spark key concepts

RDD

*Resilient*: fault-tolerant.
*Distributed*: partitioned across multiple nodes.
*Dataset*: collection of data.

Dataframes

Data organised in columns built on top of RDDs.
Better performance than RDDs.
User friendly API.
Execution on Spark

Driver Program

```python
sc = SparkContext()
spark = SparkSession(sc)
sdf = spark.read_parquet(...)
sdf.filter(...)
sdf.write_parquet(...)
```

Spark Context

DAG Scheduler

- RDD1
- RDD2
- Task
- Task
- Task

Task Scheduler

- Tasks

Cluster Manager

Worker Node

- Executor
  - Task
  - Task

Worker Node

- Executor
  - Task
  - Task

Worker Node

- Executor
  - Task
  - Task
class GenerateSmarketsAccountReport(PySparkTask, AccountingTask):

    def requires(self) -> luigi.Task:
        return GenerateAccountingReport()

    def main(self, sc: pyspark.SparkContext) -> None:
        spark = pyspark.sql.SparkSession(sc)
        sdf_per_account = read_parquet(spark, self.input())
        sdf_smarkets = sdf_per_account.filter(
            sdf_per_account.account_id == SMARKETS_ACCOUNT_ID
        )
        write_parquet(sdf_smarkets, self.output())

    def output(self) -> luigi.Target:
        return self.get_target(
            path='data/reports/accounting-report-smarkets.parquet'
        )
Scalability

- Spark cluster.
- Fast deployment.
- Easy to use.
- Flexible.
- Seamless integration with S3 - EMRFS.
- Ability to shutdown the cluster when job is done without data loss.
- Low cost.
- Nice web interface.
Spark on EMR

EMR

Master Node

YARN Resource Manager

Slave Node(s)

YARN Container

YARN Container

YARN Container
Spark on EMR

EMR

Master Node

YARN Resource Manager

Slave Node(s)

YARN Container

YARN Container

YARN Container

Client

1
Spark on EMR

EMR

Client → 1 → Master Node → YARN Resource Manager → 2 → Slave Node(s) → Spark Driver → YARN Container

1. Client initiates the process.
2. The process flows through the Master Node to the YARN Resource Manager, then to the Spark Driver, and finally to the YARN Container(s).
Spark on EMR

- **Client**
- **Master Node**
  - YARN Resource Manager
- **Slave Node(s)**
  - Spark Context
  - Spark Driver
  - YARN Container
  - YARN Container
  - YARN Container

1. Connection from Client to Master Node
2. Communication between Master Node and Spark Driver
3. Interaction between Spark Driver and YARN Container
Spark on EMR

1. Client communicates with Master Node.
2. Master Node sends tasks to Spark Driver.

EMR

Master Node

YARN Resource Manager

Slave Node(s)

Spark Driver

Spark Context

YARN Container

Spark Executors

YARN Container

YARN Container
Spark on EMR

1. Client connects to the YARN Resource Manager.
2. The Master Node sends the Spark Context to the Spark Driver.
3. The Spark Driver initializes the YARN Containers on the Slave Node(s).
4. The Spark Executors run in the YARN Containers.
5. The Spark Driver communicates with the Spark Executors through the YARN Containers.
Spark on EMR

EMR

Master Node
- YARN Resource Manager

Slave Node(s)
- Spark Context
- Spark Driver
- YARN Container
- Spark Executors

Client

1. Communication between Client and YARN Resource Manager
2. YARN Resource Manager communicates with Spark Driver
3. Spark Driver starts YARN Container
4. YARN Container communicates with Spark Executors
5. Spark Context communicates with Spark Driver
6. Spark Executors communicate with YARN Container
Thanks!
```python
class FetchMemberDetails(AccountingTask):
    def run(self) -> None:
        user_service_client = UserServiceClient()
        members = user_service_client.get_members()
        df_member_details = pd.DataFrame.from_records(members)

        with self.output().open('w') as file_:
            df_member_details.to_parquet(file_, engine='pyarrow', compression='SNAPPY', flavor='spark')

    def output(self) -> AccountingTarget:
        return self.get_target(path='data/raw/member-details.parquet')

class S3DirectoryTarget(Target):
    @contextmanager
    def operate(self, mode: str) -> Generator[str, None, None]:
        if mode not in ('r', 'w'):
            raise ValueError('Unsupported open mode "{}"'.format(mode))

        output_tmp_dir = os.path.join(self.output_tmp_dir, mode)
        pathlib.Path(output_tmp_dir).parent.mkdir(parents=True, exist_ok=True)

        if mode == 'w':
            yield output_tmp_dir
            self.client.put_dir(local_dir=output_tmp_dir, destination_s3_dir=self.s3_dir, flag=self.flag)
        elif mode == 'r':
            self.client.get_dir(s3_dir=self.s3_dir, destination_local_dir=output_tmp_dir, flag=self.flag)
            yield output_tmp_dir

    def exists(self) -> bool:
        return self.client.exists(self.s3_dir)
```
Submit Spark application to EMR from Luigi

Luigi task
- Create Spark cluster on EMR
- Upload task pickle to S3
- Add step to EMR cluster to execute application script
- Poll for step status

Application script
- Unpickle Luigi task
- Create SparkContext instance
- Run Luigi task main()

Luigi task main()
- Dataframe and RDD operations
- Store result in S3
- A task won’t raise an event if one dependency has failed.

- In case of a dependency failure, we want to destroy cluster if the only tasks left depend on failing task.

- Information about pending tasks and task dependencies fetched from Luigi Central Scheduler.