leboncoin

Eventually Consistent Data Operations

With Apache Airflow

By Nicolas Goll-Perrier
@ngollperrier

2019-02-06

Summary

28,1 million
unique visitors*

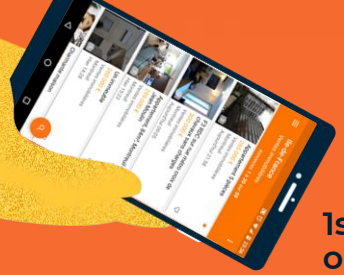
more than
27 million
classifield ads online

800 000
new ads every day

73
categories

*Source Médiamétrie Net Ratings, avril 2018

leboncoin



1st french website on the top 10 on audience

50,6 Google

45,9 facebook.

44,8 YouTube

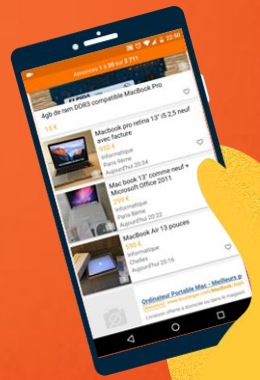
29,5 WIKIPÉDIA

28,2 leboncoin

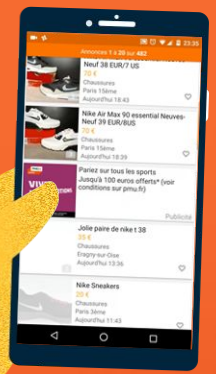
27,7 amazon

An app (iOS + Android)
downloaded 27 million times

70% of the audience
is on device mobile

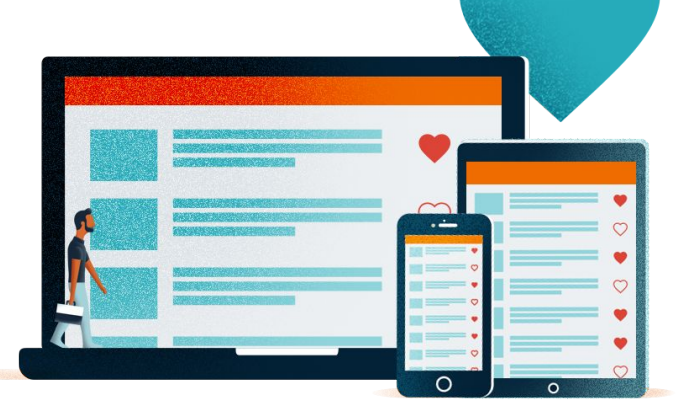


The satisfaction rate of active users 85%**



*Source Médiamétrie Net Ratings, avril 2018
**Source Baromètre de satisfaction BVA novembre 2017

leboncoin



Technical stack

2
Datacenters
&
1
Cloud provider

2000
Virtual machines

15 Gbits/s
outflows & a database of
6 To

300M
images
36k
req/s on leboncoin

Tech team of more than
200
people

A strong « open-source » culture:
PostgreSQL, Spark, Kafka, Python, Go, Kubernetes
...

Summary

A typical (good) day of data operations

Recent Tasks	Last Run	DAG Runs
41	2019-01-28 00:00	43
114	2019-01-29 10:00	272
5	2019-01-28 00:00	344
26	2019-01-28 00:00	607
105	2019-01-29 10:10	274
6	2019-01-29 09:04	177
24	2019-01-29 05:32	72
9	2019-01-28 00:00	66
12	2019-01-28 00:00	607
24	2019-01-29 08:19	269
16	2019-01-28 00:00	113

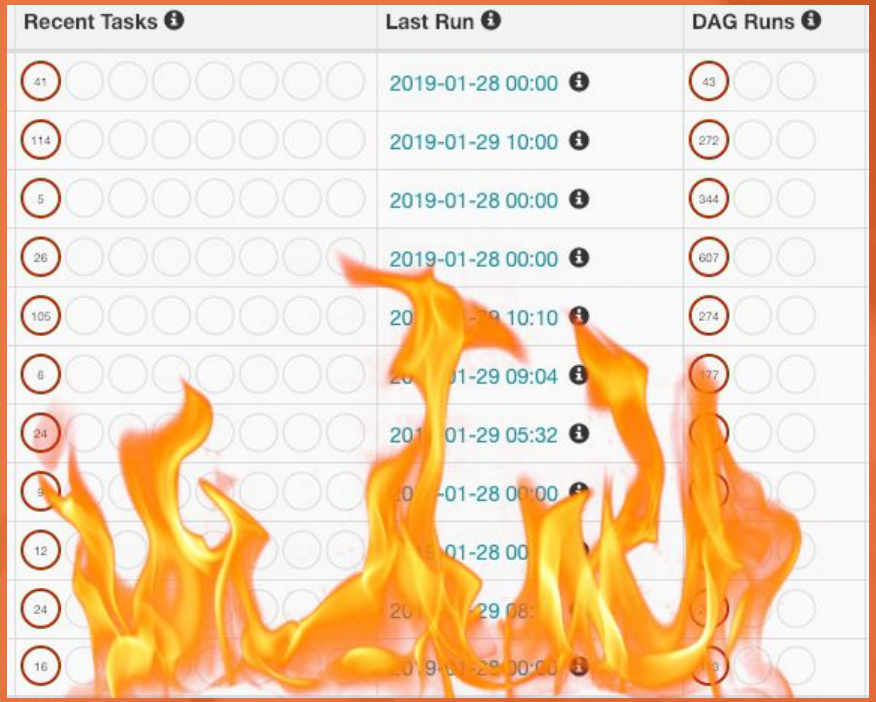
leboncoin

A typical (bad) day of data operations

Recent Tasks	Last Run	DAG Runs
41	2019-01-28 00:00	43
114	2019-01-29 10:00	272
5	2019-01-28 00:00	344
26	2019-01-28 00:00	607
105	2019-01-29 10:10	274
6	2019-01-29 09:04	177
24	2019-01-29 05:32	72
9	2019-01-28 00:00	66
12	2019-01-28 00:00	607
24	2019-01-29 08:19	269
16	2019-01-28 00:00	113

leboncoin

A typical (memorable) day of data operations



leboncoin

What could possibly go wrong ?

Things that can fail loudly:

- **Access to source data**
  - Connectivity issue to a source database
  - Maintenance on source database
  - Missing log files, partitions
  - Rate limiting or parallelism quotas reached
  - Third party API down
- **Transformation and storage**
  - Resource allocation failure (memory, network, I/O, storage space)
  - Unplanned dataset size
  - Malformed/Unexpected fields in dataset
  - Incomplete test coverage leading to crashes
  - Inadequate capacity planning

Potential impacts:

- **Missed SLA**
- **Business Reactivity delay**
- **Cost of failure investigation**
- **Outdated ML models**
- **Catch-up period**
- **Lack of sleep**
- **...**

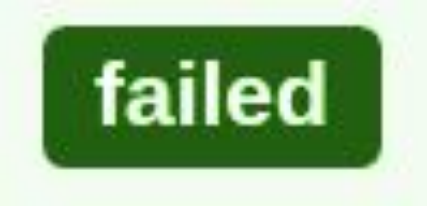
Everything is green, nothing to do ?

Things that can fail silently:

- **Product changes**
  - Unexpected business rule change
  - Changes in user behavior
- **Silent failures**
  - Success after several retries
  - Due to bugs in the code-base
  - Due to unhandled business rules
- **Inconsistent data**
  - Incomplete extraction range targeting
  - Late arriving events/facts

Potential impacts:

- **Erroneous Business Decisions**
- **Diminution of trust in KPIs**
- **Long investigations**
- **Patches will most likely be required**
- **Data re-processing**
- **...**

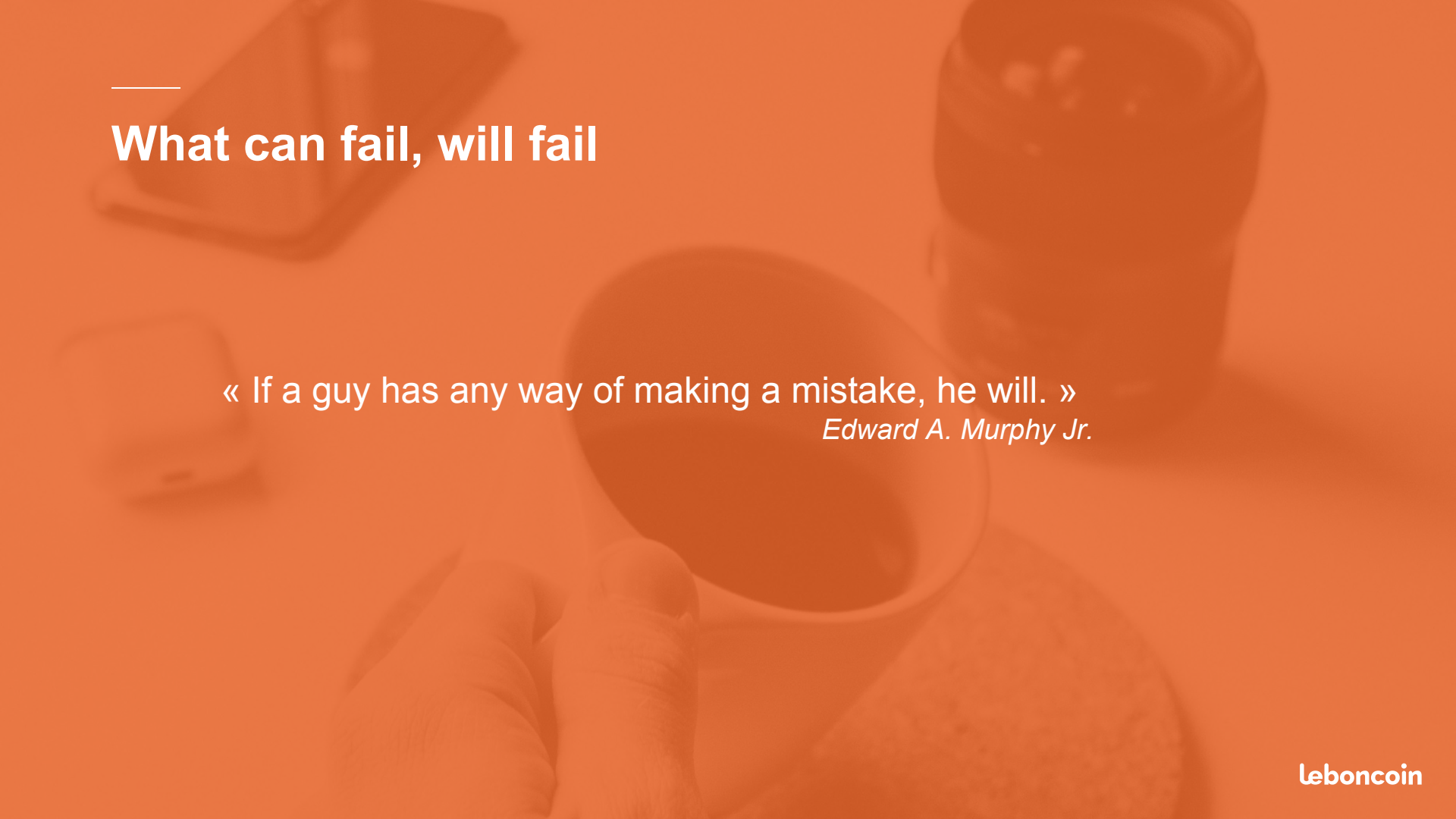


Summary

What can fail, will fail

« If a guy has any way of making a mistake, he will. »

Edward A. Murphy Jr.



leboncoin

Preventing the fire

Adequate code quality and functionalities

- Unit testing, TDD
- Strong User Acceptance Testing
- Validation of Business Rules

Detecting failures

- Thresholds and Smoke Detection Tests
- Monitoring
- Communication

But…

- Not everything can be realistically planned for
- Data systems are downstream the information system
- Upstream errors may require time and full collaboration of many other teams

Hope for the best, plan for the worst

Recovery process should focus on:

- Identifying the origin of the issue (through airflow logs)
- Communicating on the impacts of the failure
- Fixing the cause of the issue
- Or waiting for an adequate fix
- Re-running the affected task instance after resolution
- Validate and communicate the availability and accuracy of the recovered data

Hope for the best, plan for the worst

Recovery process should NOT:

- Block independent or unaffected task instances
- Require large re-computation of unaffected dag runs
- Require Business specific knowledge
- Require manual data-cleanup or ninja patching of datasets
- Risk corrupting datasets/metrics

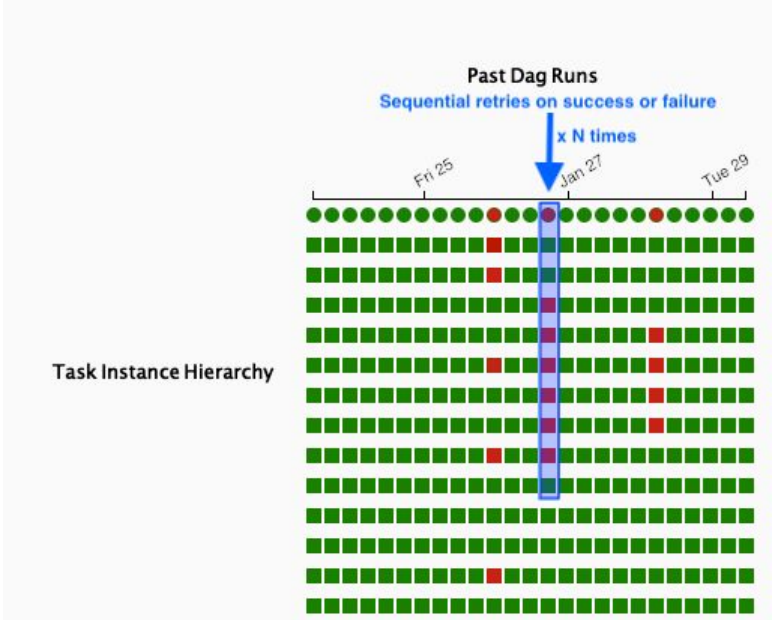
Summary

Providing replay agility

Idempotence

- Allows running of task instance any number of times
- Deterministic
- Typical use cases:
  - Re-running after patching source data
  - Re-running after fixing a bug
  - Re-running “just to be sure”

$$f(f(x)) = f(x)$$

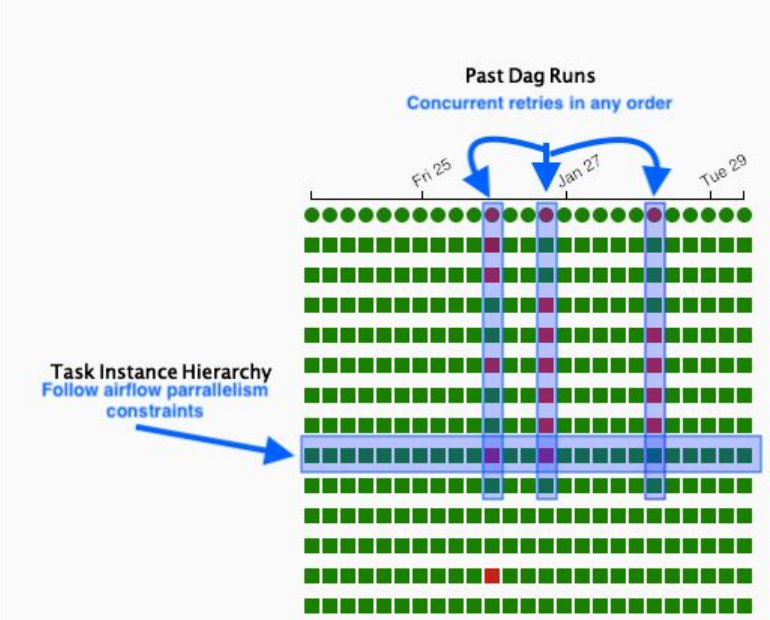


Providing replay agility (contd.)

Commutativity

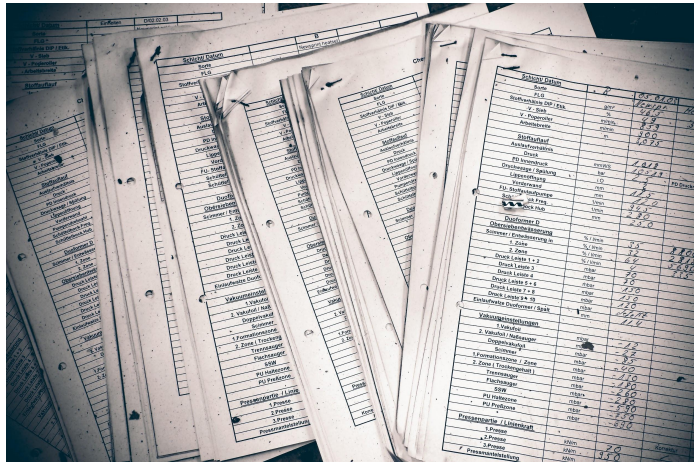
- Allows running of multiple instances of the same task in any order
- Typical use cases:
  - Non-blocking failures
  - Parallelizing re-run without a race condition

$$f(g(x)) = g(f(x))$$



Types of data objects

- **Fixed/Predictable objects**
  - Should never change during the lifetime of an application
  - Trivial, but very rare
  - Example: Time…
- **Immutable objects**
  - Once produced, cannot be altered
  - Low complexity
  - Availability depends on your information system
  - Examples: Stream of events, Logs, Business Facts (bills)
- **Mutable objects**
  - Have their own lifecycle, can change any number of time
  - Medium to High complexity (often nested)
  - Changes should be traced either by source application or change data capture
  - Examples: Domain objects (ads), Reference data (categories)…



Types of data objects (contd.)

Atomic datum

- As fine grained as can be
- Have their own life-cycle
- Can disappear over time (GDPR)

Aggregated data

- Often time-bound
- Inherit their properties from the underlying datum types
- Should converge towards the truth and be frozen

Identification and Versionning

- **Datum Identification**
  - Use natural keys whenever possible
  - Technical keys otherwise (sequence, event-id, UUID, …)
- **Dataset/batch Identification**
  - Depends on extraction mechanism
  - Generally the execution_date is a very good candidate
- **Datum Versioning**
  - Version input/output schemas for every unit of work
  - Include code version in processed output rows
  - Include actual processing date in each output rows

Atomic units of work

Atomic Individual Tasks

- Low coupling between tasks of different kind
- Single unit of work, don’t over-stretch your operations over multiple tasks
- Isolate business rule changes from technical changes
- Freeze outputs of all types of tasks on HDFS/Cloud Object Store
- Avoid past task instance/dag run dependency
- Stateless tasks if possible (don’t overuse XCOMs)

Summary

Replayable data extraction

Mostly outside of your scope...

- Bound by transactional system choices
- Many competing storage technologies
- Each comes with its own constraints and benefits
- Tuned for transactional performance
- Rarely exhaustive representation of the domain data

Or is it ?

- Be aware of the schema exposed (tables, events, APIs)
- Be aware of the objects life-cycle in your source system
- Communicate with your back-end engineering team to converge towards a mutually beneficial solution

Replayable data extraction

Detect changed objects in your source system

- Via application level mechanisms (preferably)
- Via Change Data Capture systems
- Via Snapshots (as a last resort, and not always possible)

Wait for completeness of your data-batches

- Identify key window bounds for your delta extracts
- Immutable can be incomplete too: plan for late arriving events

Avoid extraction replays as much as possible

- Store and freeze your unaltered extracts on HDFS/Cloud Object Store
- Following the guidelines above, you should rarely need to replay extractions
- Focus on resilience rather than replayability
- Most of the time, you won't be able to guarantee deterministic behavior anyway

Deterministic data transformations

Don’t couple extract and transform

- Extracted data should be persisted as closely to its original form as possible
- Extracted data should normally never be altered

The “easy” part, right ?

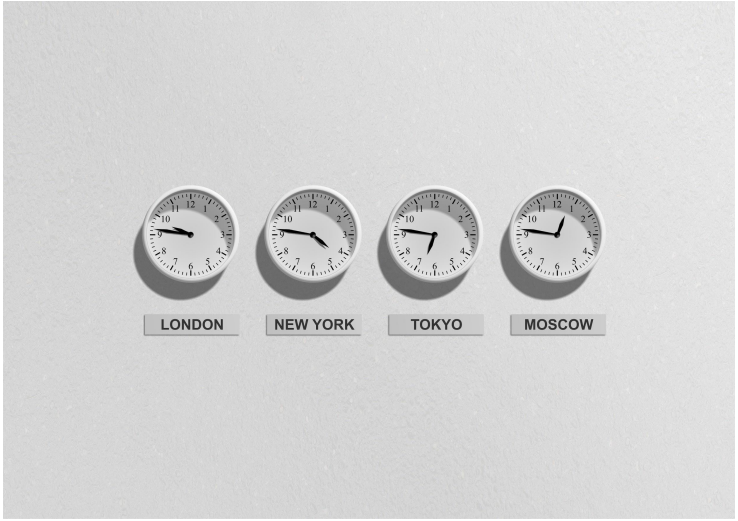
- Stateless: No external moving parts
- Can easily be isolated and tested in a sandbox
- Commutativity and Idempotence are inherently guaranteed

Single or Multiple Input => Single Output

Deterministic data transformations

Not quite as “easy” as it seems

- Identify whether business Rules are retro-active or not
- Ensure backward-compatibility for retro-active business rules
- Implement non retro-active business rules versioning either in code or in a separate configurable dataset



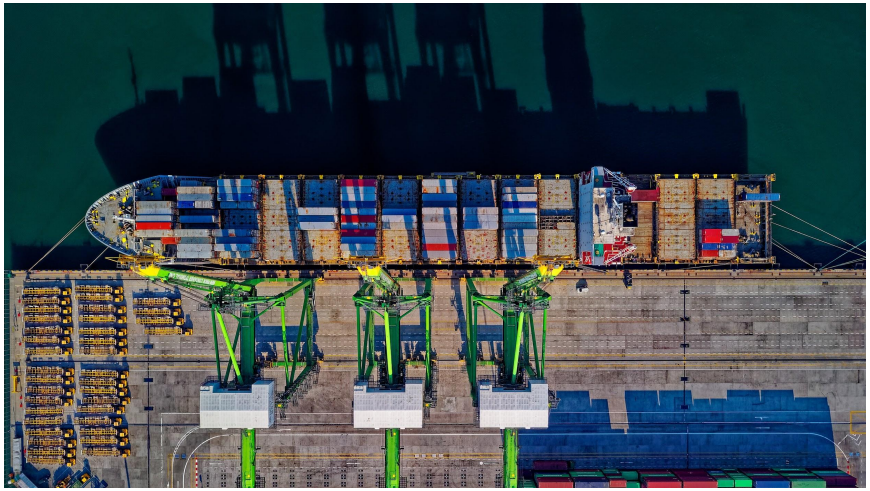
Eventually consistent data loading

Highly dependant on datastore engine

- Optimal choices may not be implementable with your storage engine
- Some might require more work than others
- Some might have degrade performance more than others

Ideally

- Overwrite, but also delete everything from the previous run (based on data version and dataset identification)
- Use fine-grained or coarse grained erasure depending on the underlying type of data objects
- Prefer ETL to ELT (whenever possible)
- Single Input => Single Output



Eventually consistent data loading

Adopt a dimension change policy:

- Reflecting your business requirements
- As consistent as need be, as simple as can be

For optimal consistency, prefer (in that order):

- Early Denormalization
- Dimension Snapshots
- Slowly Changing Dimension mechanisms
- Upsert Dimensions

Summary

Redshift Example #1

```
/* BULK DELETE AND REPLACE */
BEGIN;

DELETE FROM target_table
WHERE target_table.dataset_id = '{current_dataset}';

COPY target_table
FROM 's3://bucket/key/';

COMMIT;
```

Pros

- Idempotent
- Fast
- Simple
- Low additional I/O

Cons

- Non commutative

Ideal for

- Immutable data objects
- Order dependent tasks instances

Not suited for

- Mutable objects
- High parallelism of multiple task instances

Redshift Example #2

```sql
/* ATOMIC UPSERT */
BEGIN;

-- Insert input data in Temporary Table
COPY temp_table
FROM 's3://bucket/key';

-- Remove obsolete rows from target table
DELETE FROM target_table
USING temp_table
WHERE temp_table.primary_key_id = target_table.primary_key_id
    AND
    (
        target_table.data_version <= temp_table.data_version
        OR
        target_table.code_version <= temp_table.code_version
    )
;

-- Insert newer rows from temp table
INSERT INTO target_table
SELECT /*...*/
FROM temp_table
LEFT OUTER JOIN target_table USING (primary_key_id)
WHERE target_table.primary_key_id IS NULL;

COMMIT;
```

Pros

- Idempotent
- Commutative

Cons

- Complex
- Slower
- High additional I/O

Ideal for

- Mutable data objects, Snapshots
- Order independent, tasks instances

Not suited for

- High cardinality datasets
- Any parallelism of multiple task instances

Elasticsearch/Spark Example #1

Externally provided version

- Specify the column carrying the version
- Let ES index the new version separately

Pros

- Out of the box support in Hadoop ES
- Configuration only
- Detects version conflicts

Cons

- Fails on version conflict…
- Inadequate for retryable tasks

```
import org.apache.spark.SparkConf;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

import static org.elasticsearch.spark.sql.api.java.JavaEsSparkSQL.saveToEs;

SparkConf conf = new SparkConf(/*...*/)

conf.set("es.nodes", "elasticsearch")
    .set("es.port", "9200")
    .set("es.write.operation", "index")
    .set("es.mapping.version", "version")
    .set("es.mapping.version.type", "external");

SparkSession spark = SparkSession.builder().config(this.sparkConf).getOrCreate();
Dataset<Row> df = spark.read().format("parquet").load('s3://bucket/key');

saveToEs(df, index_mapping_name);
```

Elasticsearch/Spark Example #2

Externally provided version+conflict ignore

- Specify the column carrying the version
- Let ES index the new version separately
- Specify version conflict behavior

Pros

- Configuration only
- Detects version conflicts
- Safely discards obsolete data

Cons

- Not supported out of the box support in Hadoop ES
- Requires patch and re-compilation

```
import org.apache.spark.SparkConf;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

import static org.elasticsearch.spark.sql.api.java.JavaEsSparkSQL.saveToEs;

SparkConf conf = new SparkConf(/*...*/)

conf.set("es.nodes", "elasticsearch")
    .set("es.port", "9200")
    .set("es.write.operation", "index")
    .set("es.write.conflict.ignore", "true") // <= Will enable Commutativity and Idempotence
    .set("es.mapping.version", "data_version")
    .set("es.mapping.version.type", "external");

SparkSession spark = SparkSession.builder().config(this.sparkConf).getOrCreate();
Dataset<Row> df = spark.read().format("parquet").load('s3://bucket/key')

saveToEs(df, index_mapping_name);
```

Remaining difficulties

Cross DAG dependency tracking

- Requires manual check/awareness when replaying upstream DAGs
- Automation may become easier in the future thanks to improved data lineage?

Version aware DAG Run history

- Airflow does not persist the version of the DAG alongside the DAG Run
- A re-run will always use the most up-to date version
- Providing backward compatible DAGs is the prefered handling method for now

Conclusion

- *Know and understand your business processes*
- *Know your technological stack and its limitation*
- *Be pragmatic, not dogmatic*
- *Communicate efficiently with the information system and the business stakeholders*

Let’s keep in touch…

github.com/leboncoin

leboncoin Engineering blog

@leboncoinEng