In-Memory Computing SUMMIT | EUROPE 2019

Harnessing the power of Spark for Enterprise data engineering and analytics

Vickye Jain, Associate Principal
ZS Associates
June 4, 2019

ZS is a professional services firm that works side by side with companies to develop and deliver products that drive customer value and company results

6,000+

ZSers who are passionately committed to helping companies and their customers thrive

24 OFFICES WORLDWIDE

BANGALORE + BARCELONA + BOSTON + BUENOS AIRES + CHICAGO + EVANSTON + FRANKFURT + LONDON
LOS ANGELES + MILAN + NEW DELHI + NEW YORK + PARIS + PHILADELPHIA + PRINCETON + PUNE
SAN DIEGO + SAN FRANCISCO + SÃO PAULO + SHANGHAI + SINGAPORE + TOKYO + TORONTO + ZÜRICH

DEEP INDUSTRY EXPERTISE • LEADING-EDGE ANALYTICS • STRATEGY • TECHNOLOGY

ZS

Typical enterprise data engineering & analytics problems and solutions we deal with

Variety of data, no easy access	Scalable reporting, packaged analytics	Specialized Analytical Apps	Self-serve advanced analytics
Enterprise Data Lakes	Cloud DW/BI Solutions	Web UI + NOSQL DBs	Data-science workbenches

Example use case highlights

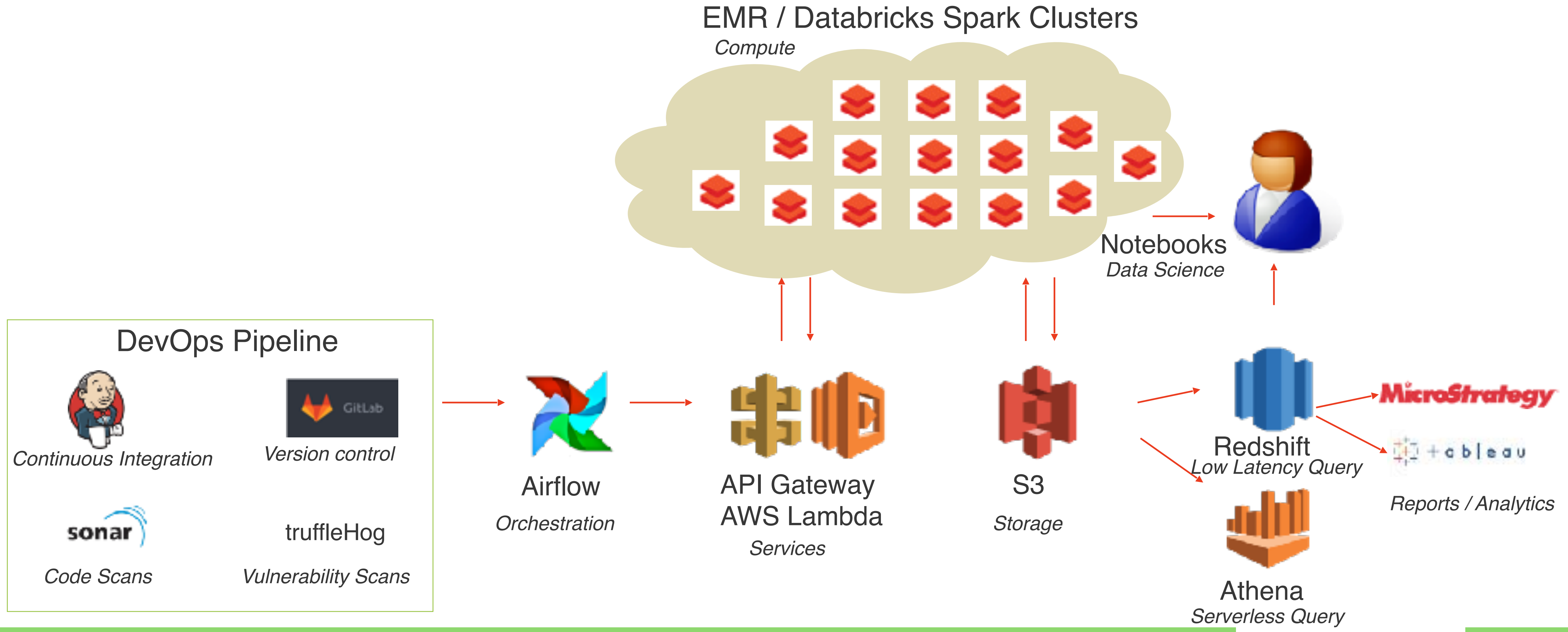
Use Case Highlights

- <24 hours SLA for Data to Reports
- 50+ data sources (S3, FTP, Internal DB, SFDC)
- 100+ analytics ready data packs
- 500+ business rules / KPIs
- 2000+ users (field + HQ)
- 3500+ GB data added weekly (500 GB inputs)

Business Challenges

- Frequently changing business rules
- Evolving internal and external input data
- Competing priorities within user group
- Complex data quality challenges
- Business and data focused internal staff

Solution Architecture



Summary of challenges

Shortfall of techno-functional experts	**Many Enterprise ETL gatekeepers have not evolved**	**Optimal infrastructure costs take some doing**	**Diversity of ETL jobs creates need for tuning**
Technical sophistication compromised when faced with tight timelines	*Scripting, CI/CD, secure SDLC, memory optimized data models, etc. need education*	*Elastic infra costs initially can be surprising, especially during Development*	*Different tuning approaches fit different job types needing continuous improvement*

SQL or Scripting?

Split application into core technical components and business logic

SQL is excellent for business logic, second nature for domain experts

Spark SQL highly optimized, will run faster in many cases

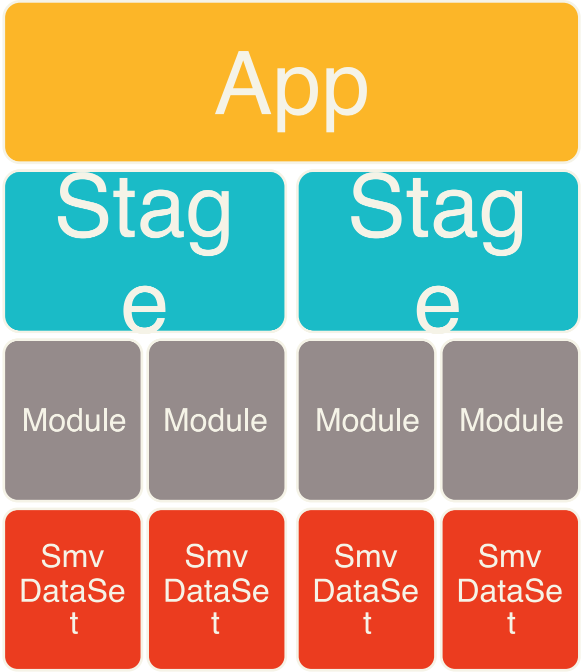
Encapsulate SQLs in PySpark shells to retain maximum flexibility

PySpark excellent for technical components, easy to read and maintain

Beauty of Spark is that both will use same execution engine and design patterns

Spark Modularized View (SMV) Data Application Framework

Enforced modularization	Without SMV:	With SMV:	Key Benefits




```
CREATE TABLE cohort AS
    SELECT DISTINCT p_id from (
        SELECT DISTINCT p_id FROM
Rx
        UNION ALL
        SELECT DISTINCT p_id FROM
Px)
```

```
class PatientCohort (SmvModule):
    def requiresDS(self):
        return [Rx,Px]

    def run(self, i):
        # Select distinct patient ids for RX claims
        d_rx = i[Rx].select('p_id').dropDuplicates()

        # Select distinct patient ids for PX claims
        d_px = i[Px].select('p_id').dropDuplicates()

        # Combine RX & PX and drop duplicates
        cohort =
d_rx.smvUnion(d_px).dropDuplicates()

        return cohort
```

Enforced modularization

Nifty ETL functions

Easily debug any step

Code wrapped with data

smv-run –run-app runs entire application

smv-run –s stagename runs one stage only

smv-run –m stagename.module runs one module only

df.**smvUnpivot**("Col1", "Col2", "Col3")

df.smvGroupBy("ID").**smvFillNullWithPrevValue**($"claimid".asc)("Indication")

https://github.com/TresAmigosSD/SMV

Extreme performance tips

Segregating storage and compute is a must for maximum elasticity

Shuffles write to disk, optimize data models to minimize joins and aggs

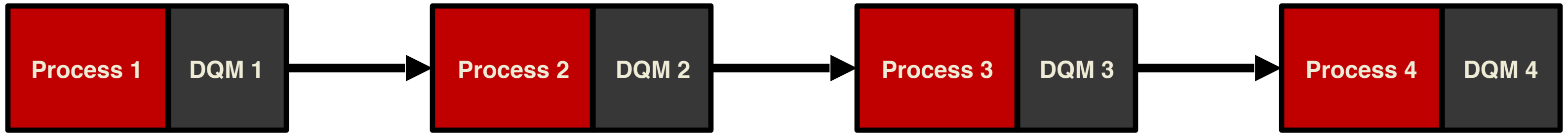
Broadcast join is your best! First thing to try for joins

Cost based optimizer is awesome! Don’t forget to analyze tables

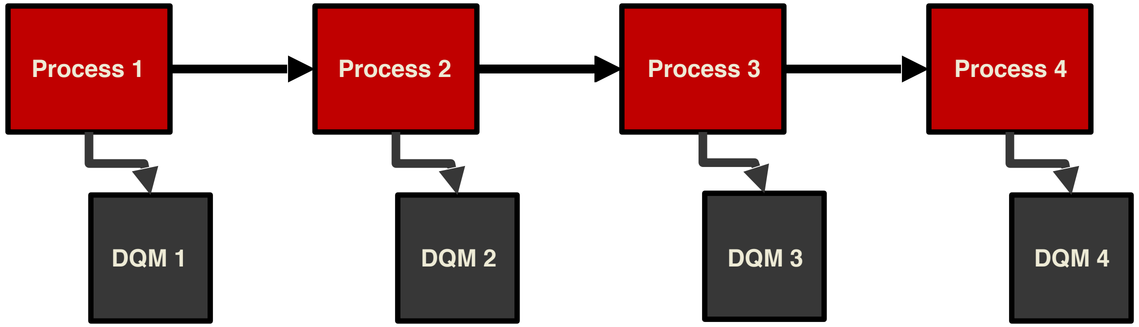
Keep **UDFs in Scala/Java**, PySpark UDFs are relatively slower

Extreme performance tips: decouple storage and compute

Process and DQM in single cluster



Process and DQM in separate cluster



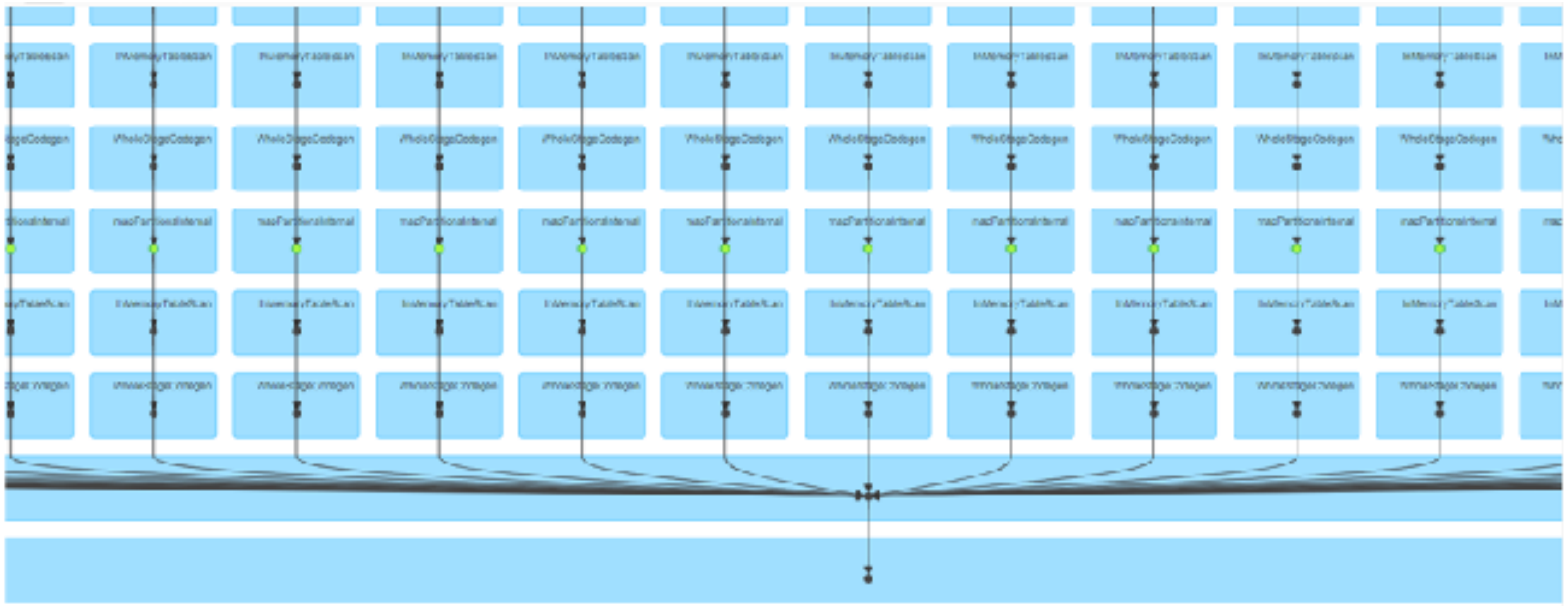
Possible only with decoupled storage and compute

**DQM – Data Quality Module*

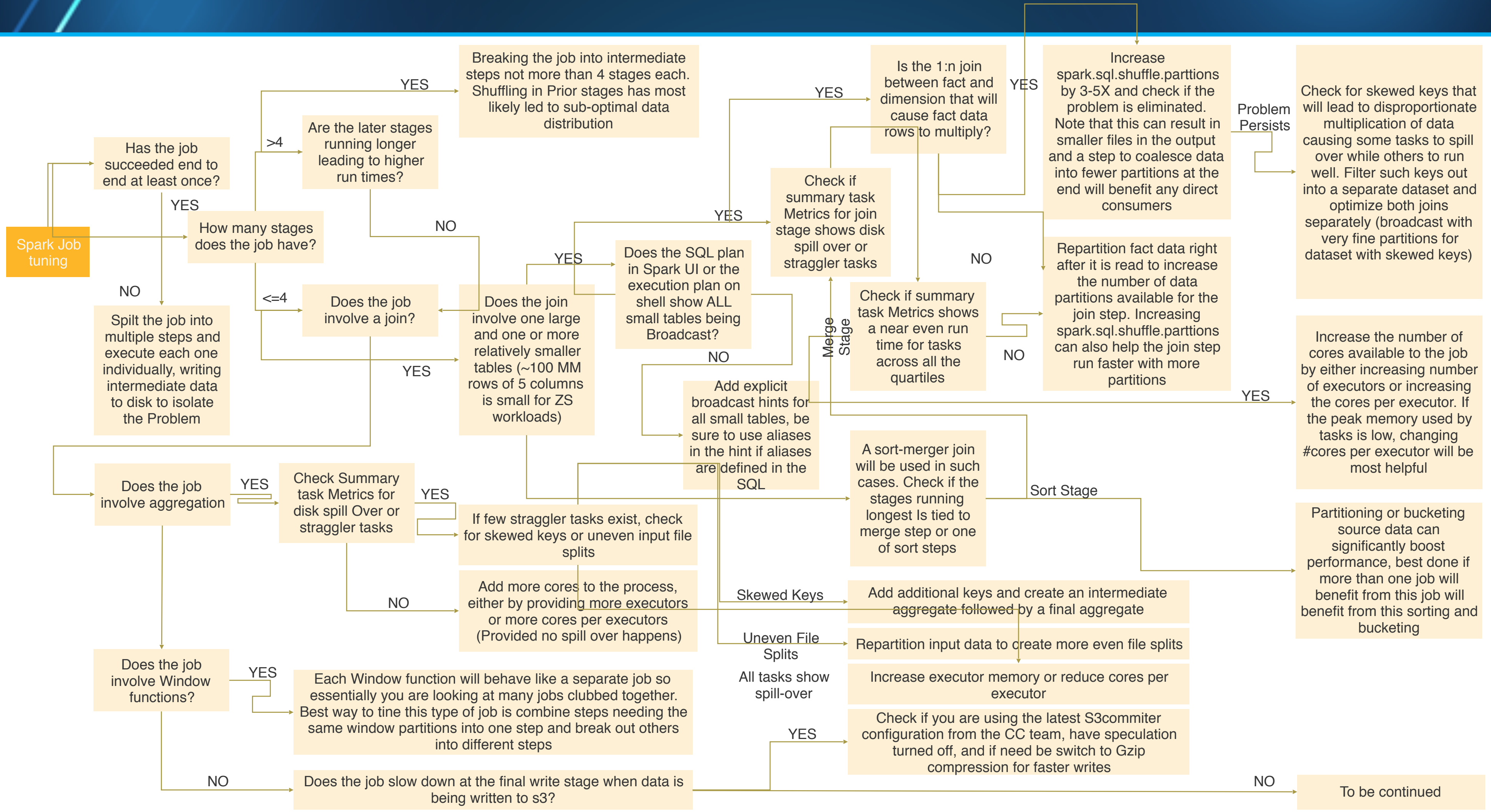
Extreme performance tips

Think of task level parallelism when packaging Spark jobs

Check 1 Check 2 Check 3 … Check n



Asking your Spark experts to codify tuning steps will also help functional experts learn to self-service



Here is an example of tuning work done by a Spark expert

Read cust level sales data (335MM) Prep dimensions for joins	*Fairly fast*	**Optimization**: Source data partitions + More cores (scales up to connection limit)
Join with Time bucket dimension (1.3B rows)	*Reasonably fast*	**Optimization**: Broadcast join + More partitions + More cores
Aggregate (332MM rows)	*Reasonably fast*	**Optimization**: More partitions + More cores
Join with Product dimension (1.7 B rows)	▪ *No broadcast* ▪ *Some disk spill-over*	**Optimization**: Broadcast join + More partitions + More memory per task / core (bigger nodes)
Aggregate & write to S3 (1.15 B rows)	*Reasonably fast*	**Optimization**: More partitions + More cores
Read back Market data (412MM)	*Reasonably fast*	**Optimization**: More cores
Replicate each market row for all products in the market and write to S3 (6.2B rows)	▪ *No broadcast* ▪ *Very large disk spill-over*	**Optimization**: Broadcast join + Large number of partitions + More memory per node

Original performance: ~1 hour on 40 nodes (160 cores, 1280 GB memory); ~1 hour on 80 nodes (320 cores, 2560 GB memory)

Revised performance: ~40 min on 20 nodes (160 cores, 1280 GB memory); ~20 min (320 cores, 2560 GB memory)

Calling many APIs in parallel, Spark can help!

 # UDF Definition

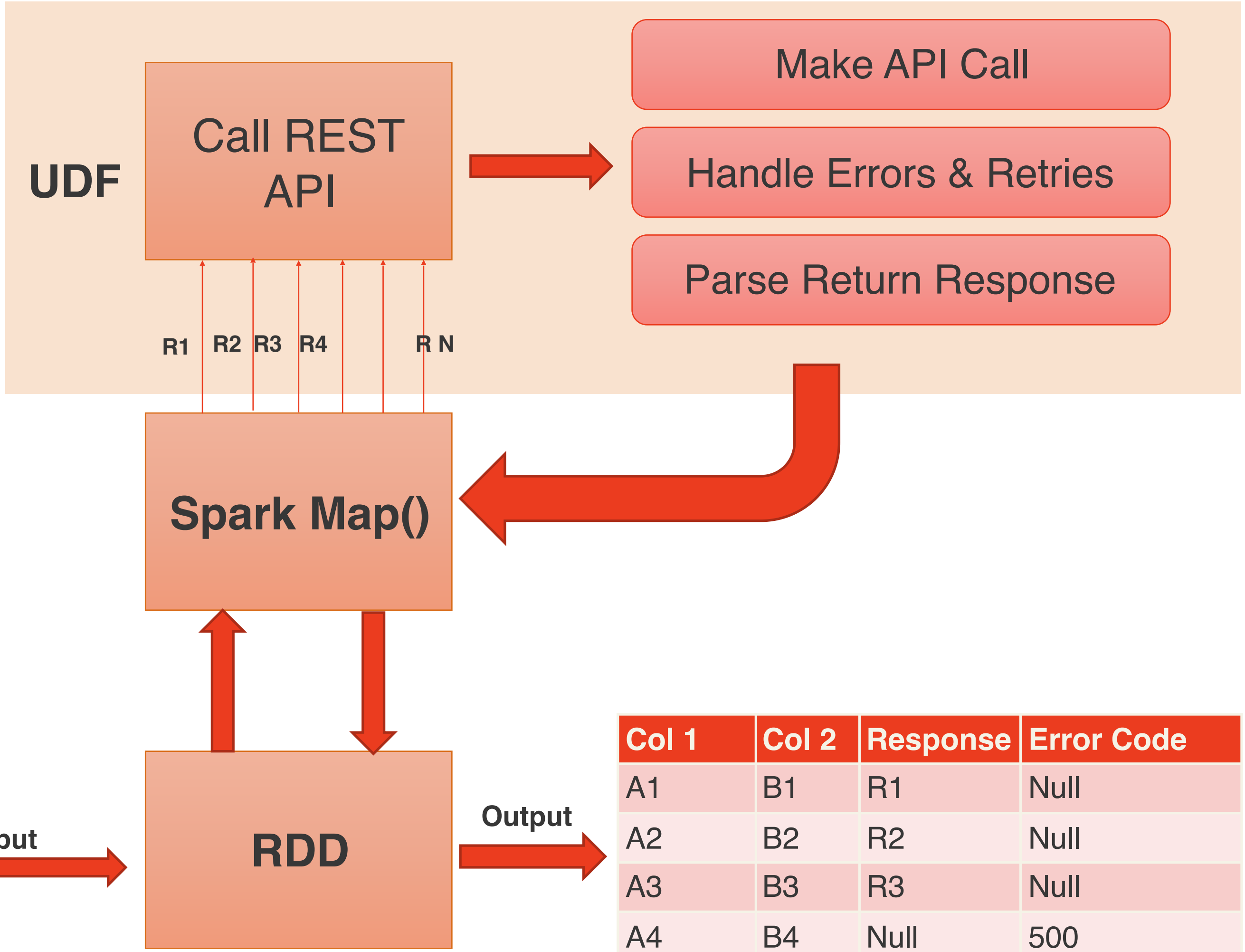
```
def api_caller(x):

    r =
post(url=url,data=json.dumps(data),headers=final_headers)
    response = r.json()['Id']
    return Row(Response= str(x[0]))
```

 # Map function to run for each df row

```
input_df = spark.sql("""select * from <table>"""

mapped_batch_df = df.rdd.map(api_caller).toDF()
```



Col 1	Col 2
A1	B1
A2	B2
A3	B3
A4	B4

Col 1	Col 2	Response	Error Code
A1	B1	R1	Null
A2	B2	R2	Null
A3	B3	R3	Null
A4	B4	Null	500

DevOps for Data Platforms

DevOps for data platforms is hard!

Rule metadata and input data change more often than code

Recommendations:

Hold data and rule metadata constant to test codes first

Create pipelines to test integrated code, rule metadata, and data together

Think of threshold based test cases rather than absolute for integration tests

Architecting for Adaptability

Mature cloud users are pivoting towards microservices architecture patterns based on AWS Lambda, AWS ECS, Docker-Kubernetes, etc.

Design modules by first defining API signatures even if not building microservices for future compatibility

Micro-APIs in AWS Lambda can easily be designed for reusability, think cluster management, job auditing, notification, partition refresh, etc.