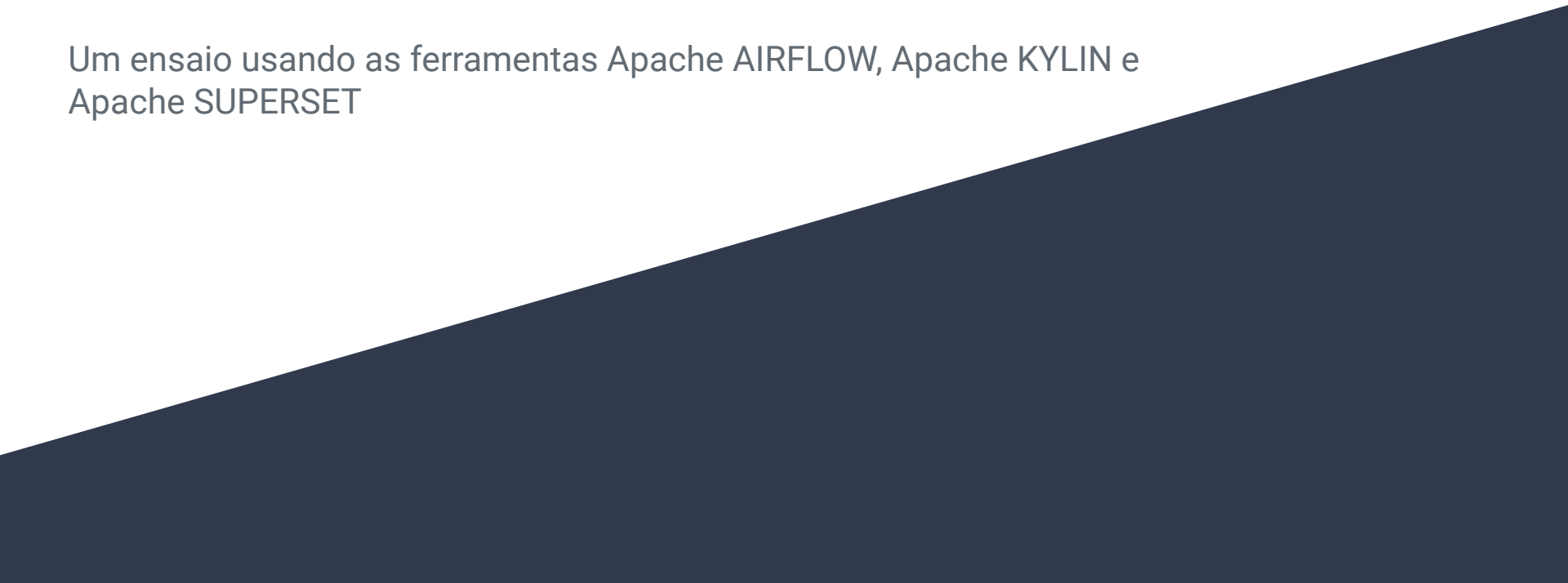


DATATEST: Elaboração e implementação de testes automáticos em volumes gigantescos de dados

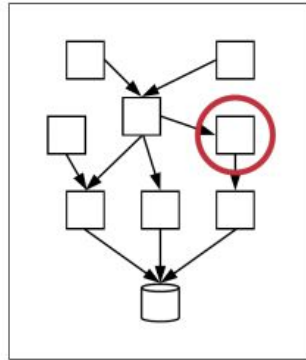
Um ensaio usando as ferramentas Apache AIRFLOW, Apache KYLIN e Apache SUPERSET

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.

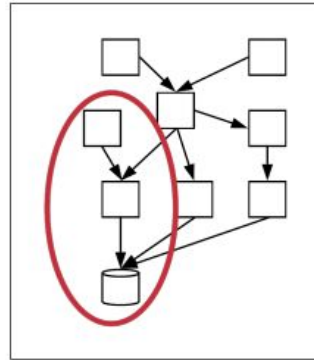
Motivação

- Testes: Manual x Automático
 - Metodologia
 - Garantias de dados limpos e corretos
- Qualidade nos **processos** do armazém (de volumes gigantescos) de **dados**
 - *Reduzir falhas prejudiciais no sistema de suporte à decisão*
- Detecção precoce de efeitos colaterais por manutenção/refatoração de ETLs
- “Lembrete” para correção de dados na origem

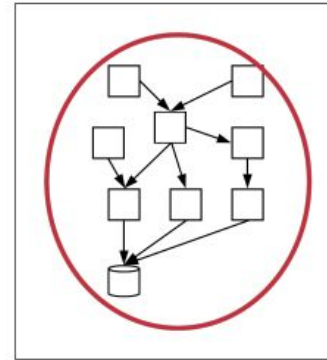
Background



Unidade



Integração



Sistema

Desempenho / Escalabilidade

Fonte: Valente, Marco Tulio. "Engenharia de Software Moderna (Livro Digital)." (2020).

<https://engsoftmoderna.info/cap8.html>

https://docs.google.com/presentation/d/1qNoRwDd1UqjymaJIUlsuvGzKF4kVeAo4fnjTTBrs4q0/edit#slide=id.g8fcf07537b_0_3

Background

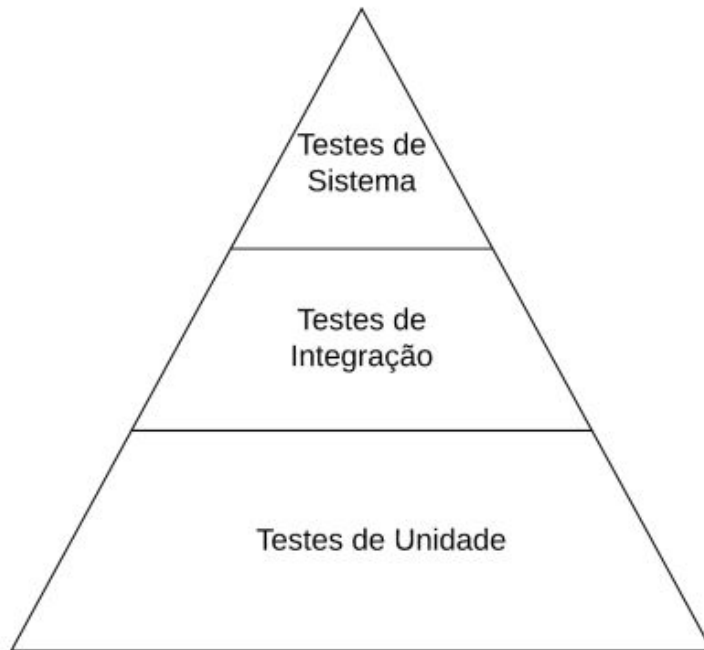
Desempenho / Escalabilidade

10%

20%

ETL

70%



↑
Maior granularidade
Menor quantidade
Mais lentos
Maior custo

↓
Menor granularidade
Maior quantidade
Mais rápidos
Menor custo

Background

Métodos de teste têm a seguinte estrutura:

- Primeiro, cria-se o contexto do teste, também chamado de **fixture**. Para isso, deve-se instanciar os objetos que se pretende testar e, se for o caso, inicializá-los. No nosso primeiro exemplo, essa parte do teste inclui apenas a criação de uma pilha de nome `stack`.
- Em seguida, o teste deve chamar um dos métodos da classe que está sendo testada. No exemplo, chamamos o método `isEmpty()` e armazenamos o seu resultado em uma variável local.
- Por fim, devemos testar se o resultado do método é aquele esperado. Para isso, deve-se usar um comando chamado **assert**. Na verdade, o JUnit oferece diversas variações de `assert`, mas todas têm o mesmo objetivo: testar se um determinado resultado é igual a um valor esperado. No exemplo, usamos `assertTrue`, que verifica se o valor passado como parâmetro é verdadeiro.

Princípios **FIRST**

- Testes de unidades devem satisfazer às seguintes propriedades:
- **F**ast (Rápidos)
- **I**ndependent (Independentes)
- **R**epeatable (Determinísticos)
- **S**elf-checking (Auto-verificáveis)
- **T**imely (Escritos o quanto antes)

Background

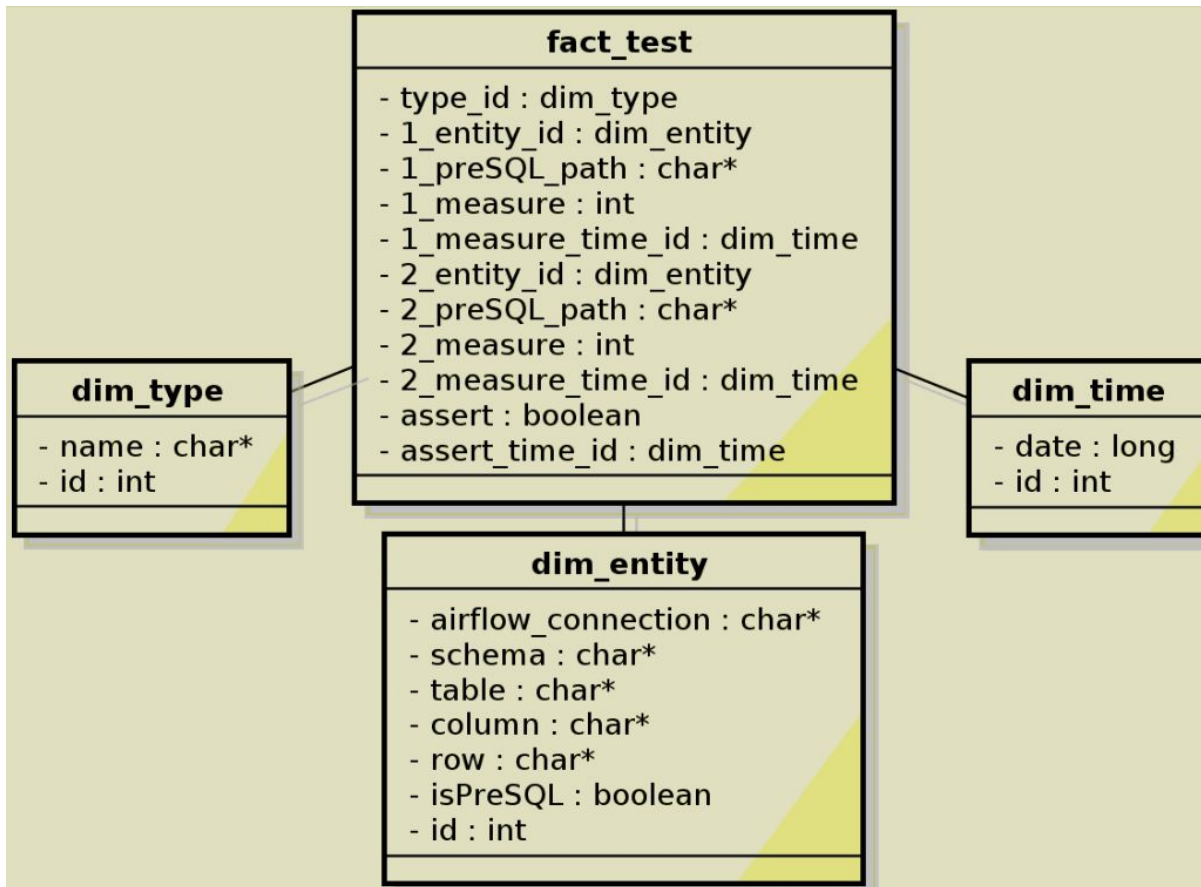
Classe de Equivalência

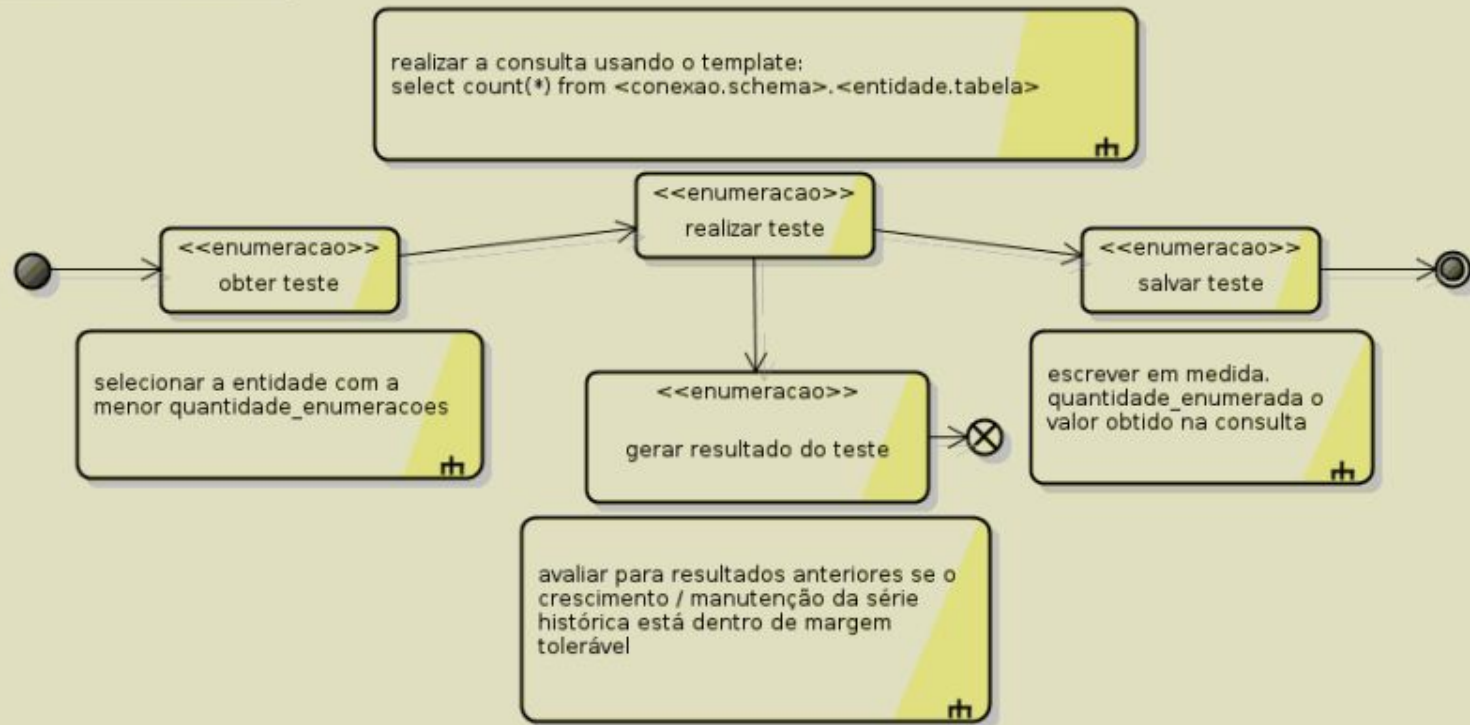
- Dividir as entradas de um problema em conjuntos de valores que têm a mesma chance de apresentar um bug
 - Conjuntos == classes de equivalência
- Para cada classe de equivalência: testar apenas um dos seus valores (pode ser escolhido randomicamente)

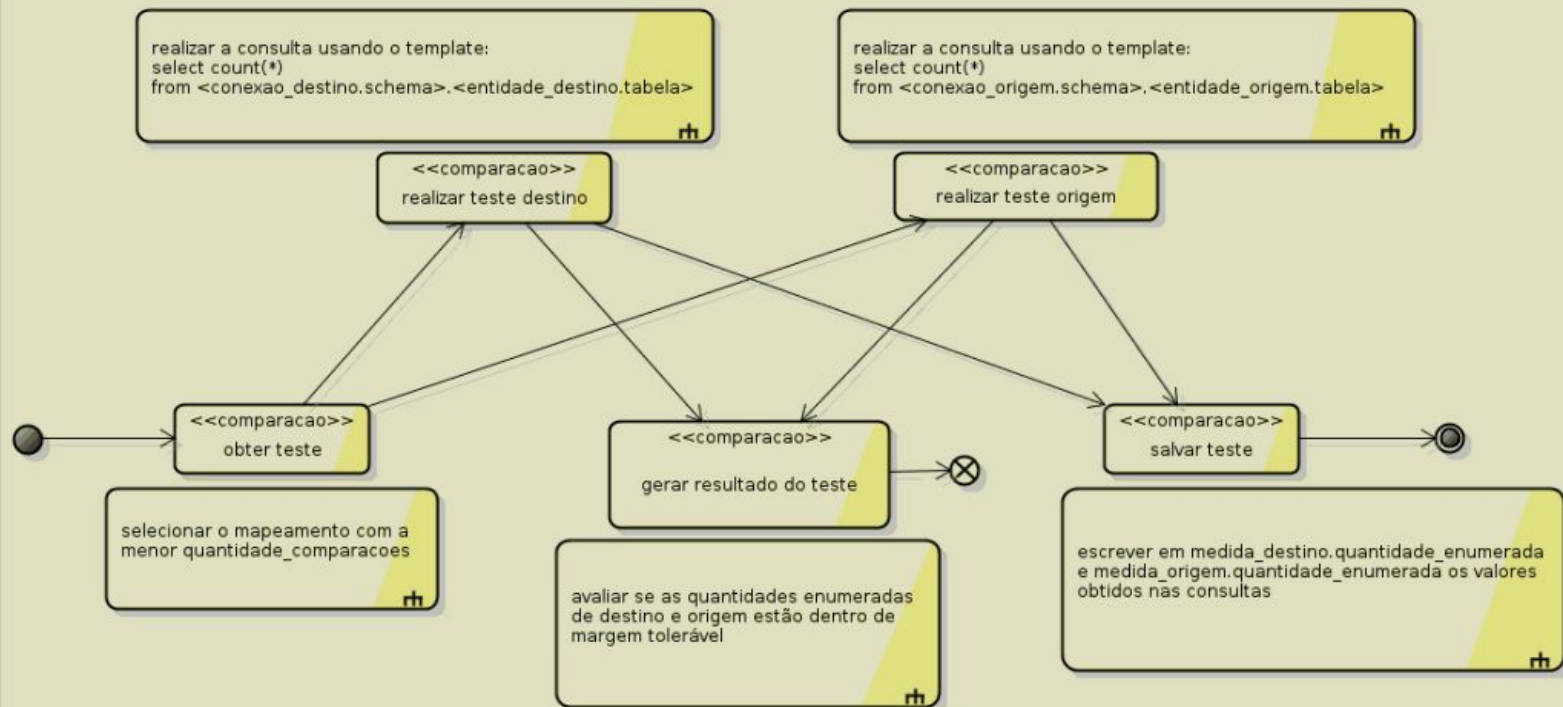
Análise de Valor Limite

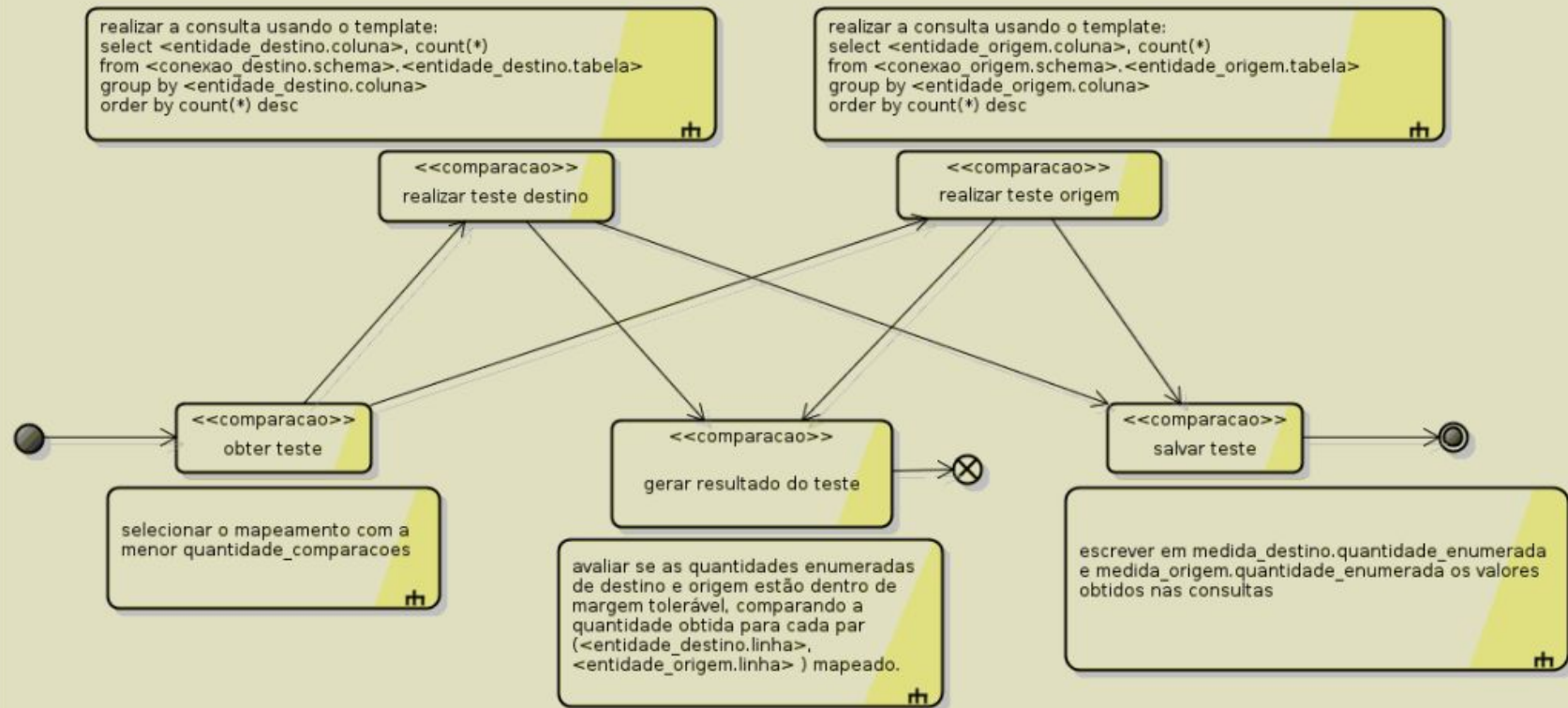
- Recomenda testar unidade com os valores limites de cada classe de equivalência e seus valores subsequentes (ou antecedentes)
- Motivo: bugs com frequência são causados por um tratamento inadequado desses valores de fronteira

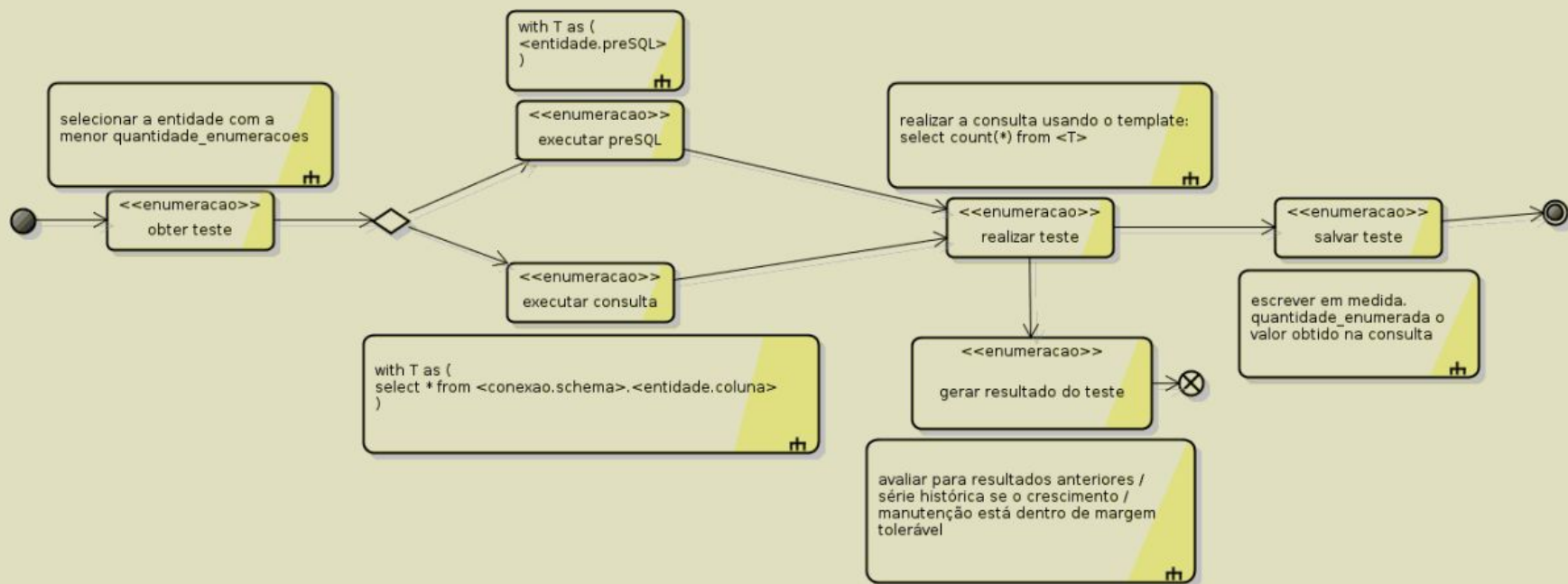
Testes e Modelo



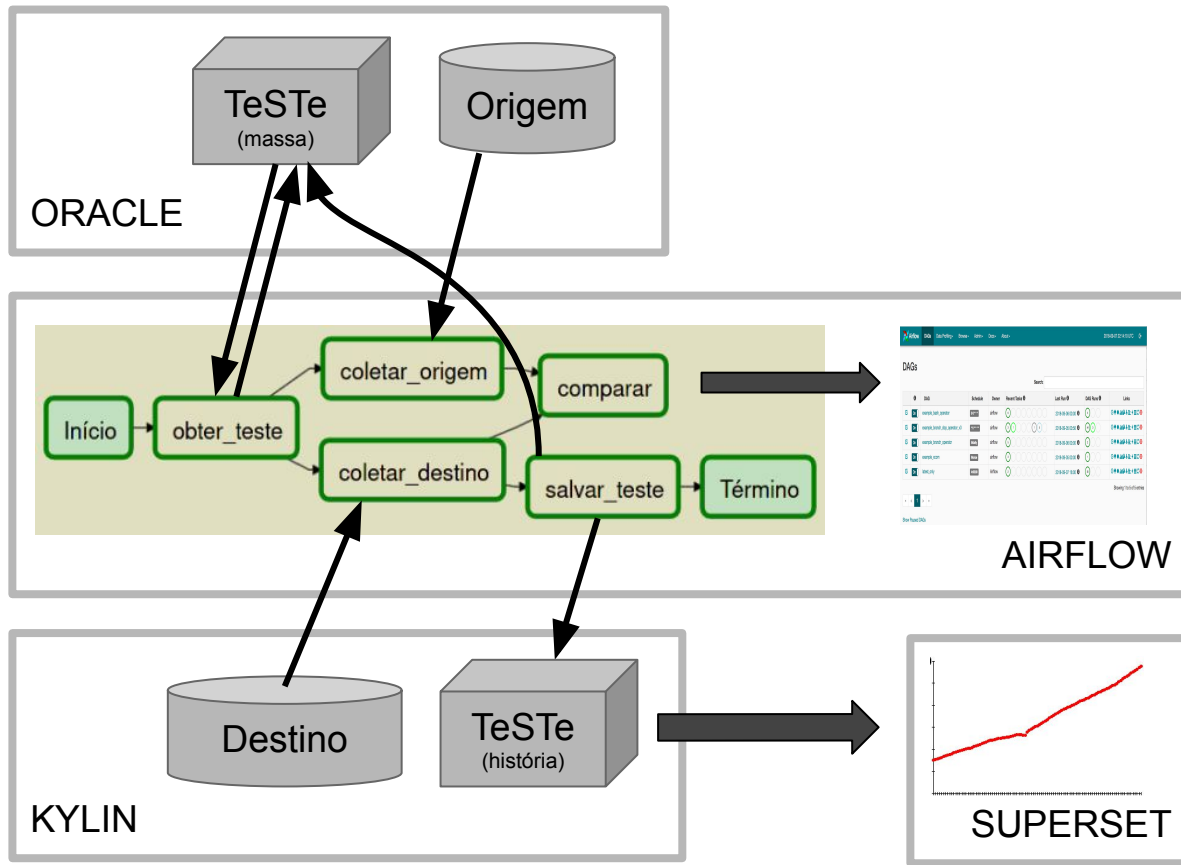




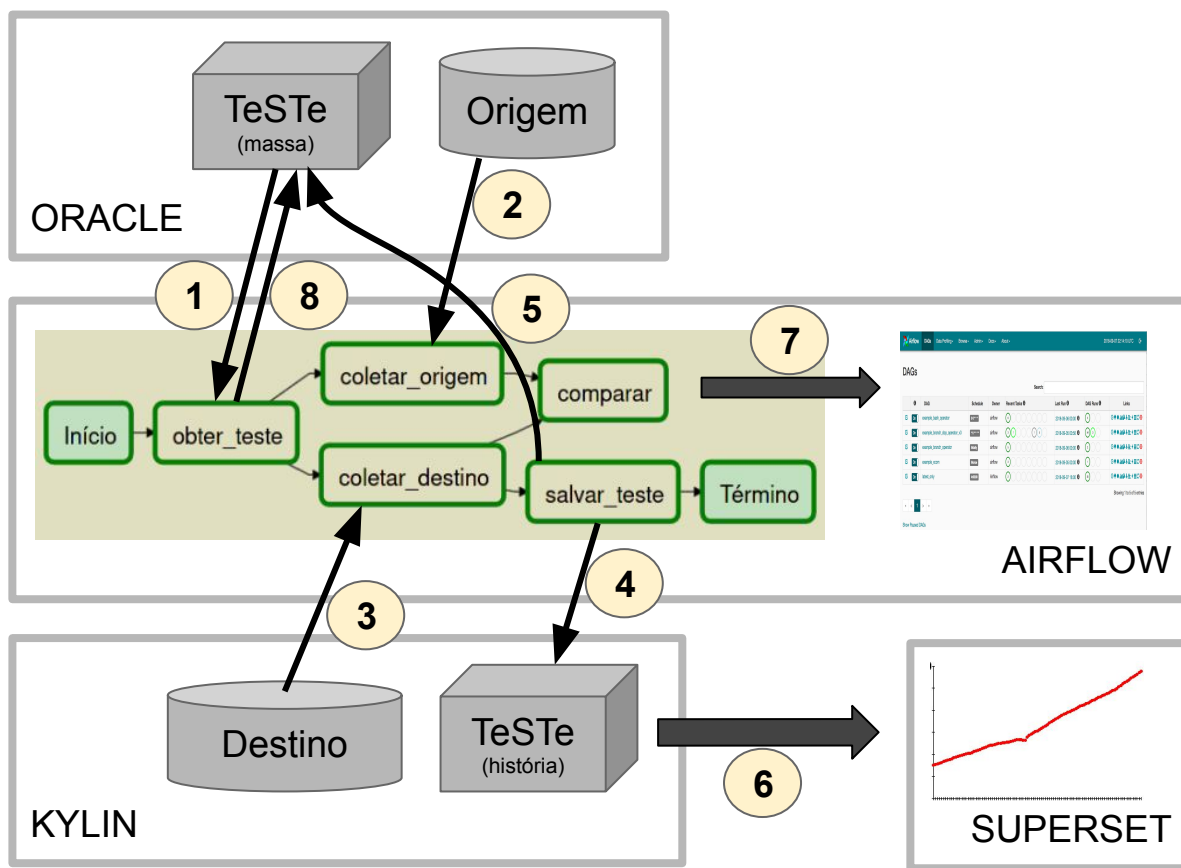




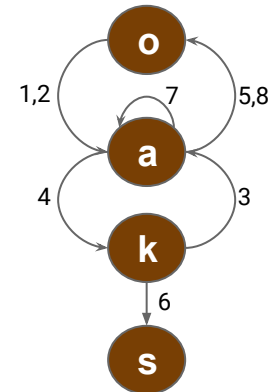
Ambiente dos ensaios



Ambiente dos ensaios

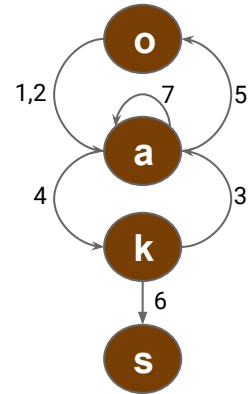


- | | |
|----------------------|---------------|
| 1) instant_client | o>a |
| 8) instant_client | a>o |
| 2) instant_client | o>a |
| 3) jdbc kylin driver | k>a |
| 4) kafka | a>k |
| 5) instant_client | a>o |
| 6) kyllnpy | k>s |
| 7) airflow_web | a>a |



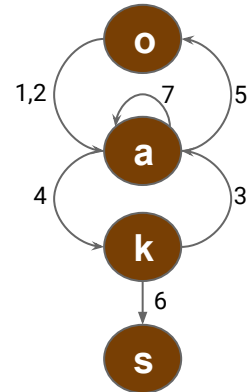
- 1) obter os dados para o teste
- 8) sinalizar que a execução do teste foi iniciada
- 2) coletar dados do teste
- 3) coletar dados do teste
- 4) salvar dados do teste (histórico)
- 5) salvar dados do teste (massa)
- 6,7) os dados do teste são visualizados

- | | |
|----------------------|---------------|
| 1) instant_client | o>a |
| 8) instant_client | a>o |
| 2) instant_client | o>a |
| 3) jdbc kylin driver | k>a |
| 4) kafka | a>k |
| 5) instant_client | a>o |
| 6) kyllinpy | k>s |
| 7) airflow_web | a>a |

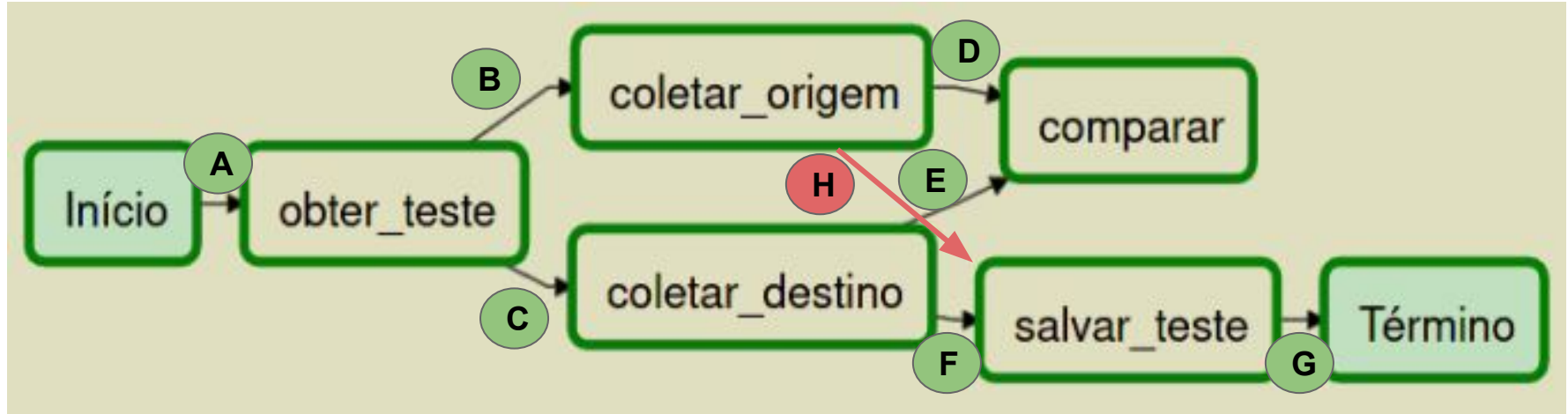


- 1) obter os dados para o teste
 - t1 construir cubo da massa (modelar no astah)
- ver 5)
- 8) sinalizar que a execução do teste foi iniciada
- 2) coletar dados do teste
- 3) coletar dados do teste
- 4) salvar dados do teste (histórico)
 - t2 construir cubo do histórico (modelar no astah)
- inserir histórico da origem
 - t3 estudar kafka do kylin_streaming_cube para aplicar no tst_cube
- 5) salvar dados do teste (massa)
 - t4 atualizar sobre finalização e resultado falha/sucesso do teste
- ver t1
- 6,7) os dados do teste são visualizados

- | | |
|----------------------|---------------|
| 1) instant_client | o>a |
| 8) instant_client | a>o |
| 2) instant_client | o>a |
| 3) jdbc kylin driver | k>a |
| 4) kafka | a>k |
| 5) instant_client | a>o |
| 6) kyllinpy | k>s |
| 7) airflow_web | a>a |



A,G) sem dependência: facilitadores de início e fim do fluxo
B,C) informações sobre conexão+tabela+colunas a serem testadas
D,E) resultados da coletas para comparação
F) resultados para histórico
H) avaliar inserir histórico da origem



Fluxo de trabalho: dependência de parâmetros

A,G) sem dependência: facilitadores de início e fim do fluxo

t5 avaliar o uso dos parâmetros json ao rodar dag na interface web

t6 avaliar agendamento de dags

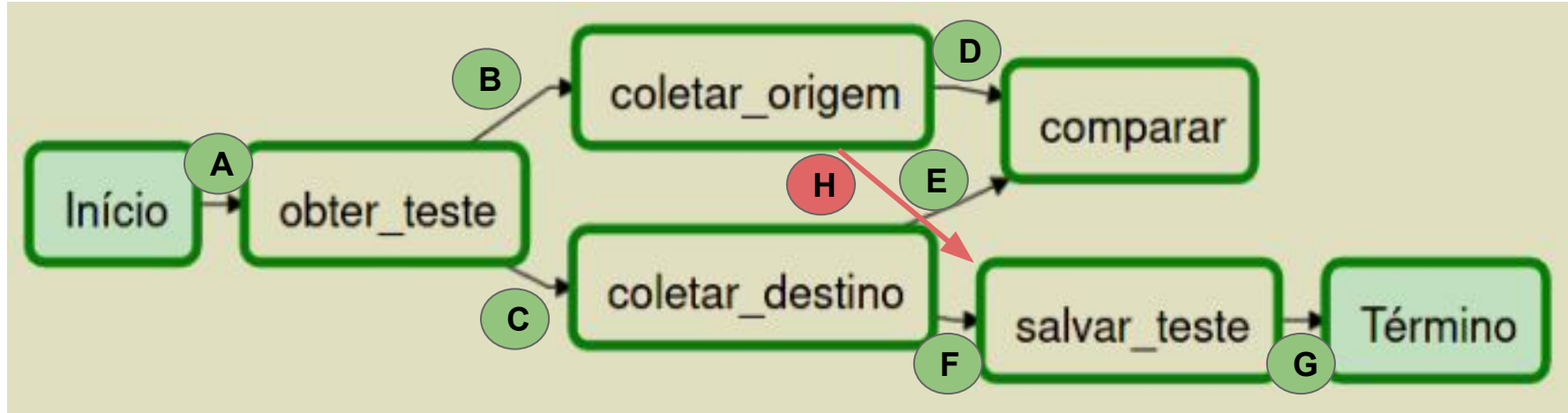
B,C) informações sobre conexão+tabela+colunas a serem testadas

D,E) resultados da coletas para comparação

t7 estudar como fazer xcom do código atual *versus* refatorar: salvarEcomparar buscarEcoletar

F) resultados para histórico

H) avaliar inserir histórico da origem



Configuração do ambiente dos ensaios

AIRFLOW

(

Postgres:13

,Redis

,Airflow-webserver

,Airflow-scheduler

,Airflow-worker

,Airflow-init

,Flower

)

localhost:8080
airflow: airflow

- Construção da imagem

git clone <https://github.com/apache/airflow.git>

cd airflow

<DOCKERFILE: edição para inserir 1) instant client Oracle e 2) driver jdbc Kylin>

docker build -t apache/airflow .

- Instanciação

curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.1.0/docker-compose.yaml'

mkdir ./dags ./logs ./plugins

echo -e "AIRFLOW_UID=\$(id -u)\nAIRFLOW_GID=0" > .env

docker-compose up airflow-init

docker-compose up

- Construção de um “fluxo de trabalho”

<DAG: *comparar_linhas*>

KYLIN

(

HADOOP

,SPARK

,HIVE

,HBASE

)

localhost:8084
ADMIN : KYLIN

- Instanciação

docker pull apachekylin/apache-kylin-standalone:4.0.0-beta

docker run -d -m 8G -p 8084:7070 apachekylin/apache-kylin-standalone:4.0.0-beta

- 1) como root setar senha (ou configurar novo usuário para ssh)

passwd

- 2) <https://stackoverflow.com/questions/21396508/yumrepo-error-all-mirror-urls-are-not-using-ftp-http-or-file>

```
vi /etc/yum.repos.d/CentOS-Base.repo mirror to vault
yum update
yum install openssh-server
service sshd start
```

- Construção de cubo

<CUBE: kylin_sales>

SUPERSET

(

Postgres:10 superset_db
,Redis superset_cache
,superset_app
,superset_worker
,superset_worker_beat
,superset_init

)

localhost:8088
admin : admin

- Construção da imagem

git clone <https://github.com/apache/superset.git>

cd superset

<DOCKERFILE: edição para inserir em requirements/local.txt> kyllinpy>

docker build -t apache/superset .

- Instanciação

docker-compose -f docker-compose-non-dev.yml up

docker exec superset_app /app/docker/docker-init.sh

- Construção de uma conexão (database/dataset)

<DATABASE -- DATASET: sqlalchemy
kylin://ADMIN:XXXXXXXXXX@<kylin_hostname>:7070/learn_kylin>

- Construção de uma visualização (chart/dashboards)

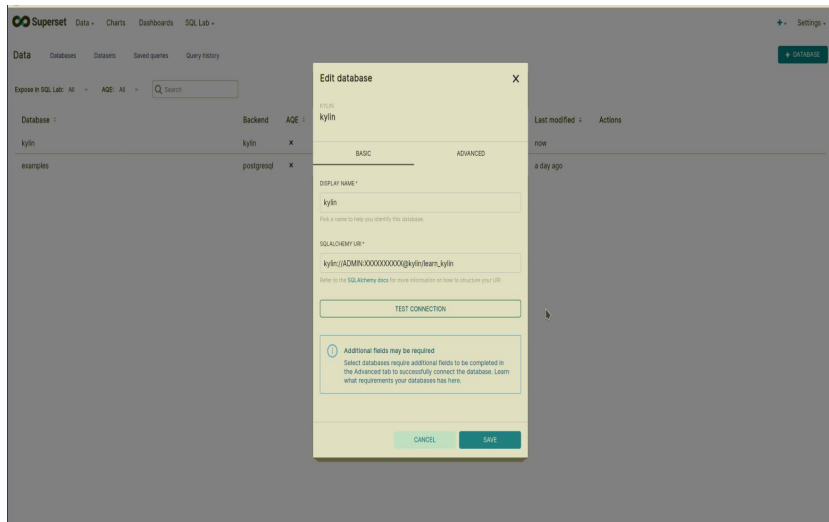
<CHART -- DASHBOARDS: TIME and QUERY>

- Utilização do sqllab (alternativa ao kylin insights?)

<SQLLAB:>

superset-kylin, airflow-oracle, airflow-kylin

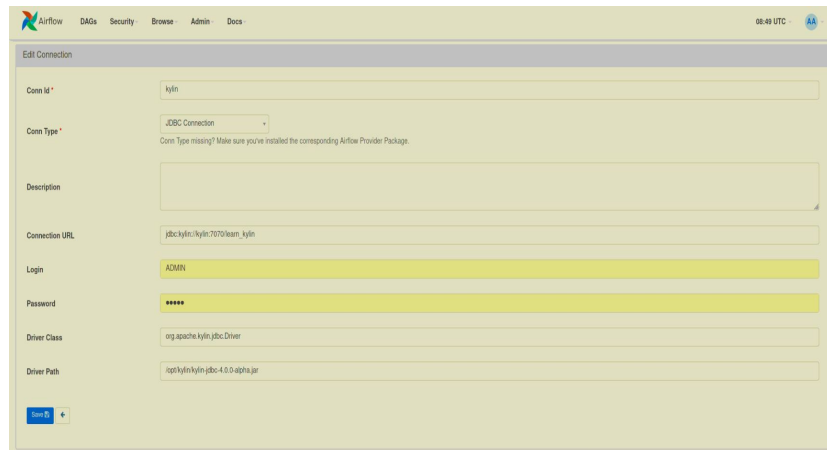
superset-kylin (kylinpy)

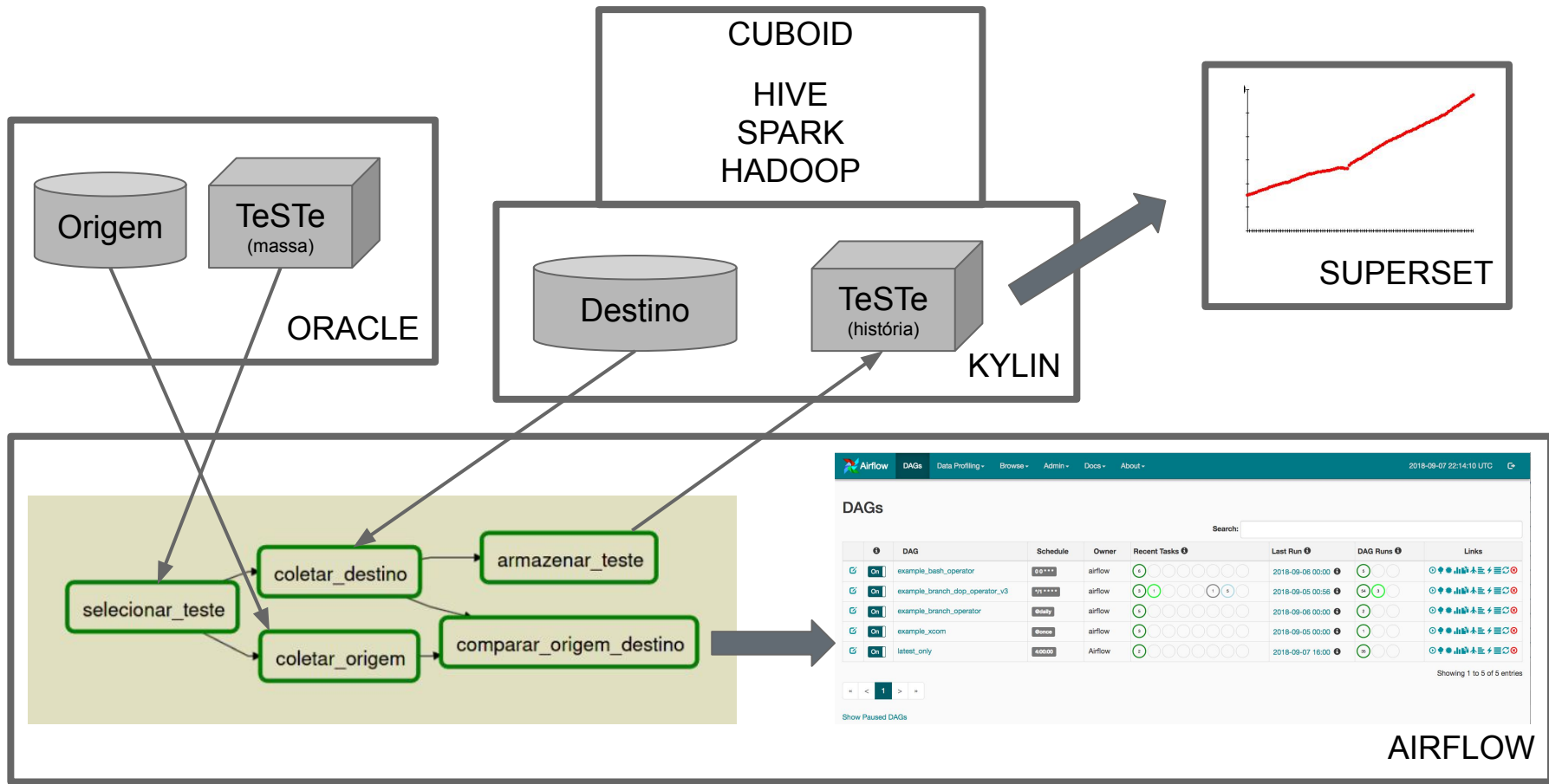


airflow-oracle (instant_client)

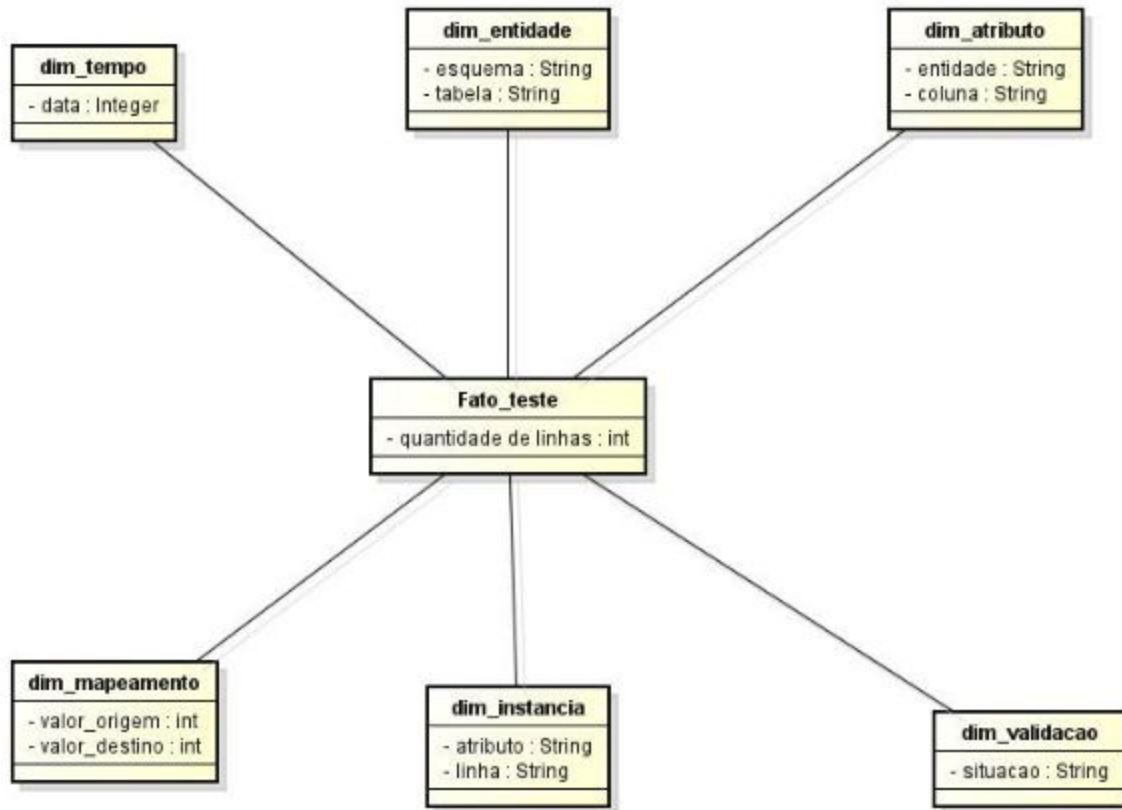
airflow connections add 'oracle' --conn-uri 'oracle://USER:PASS@HOST:PORT/SCHEMA?sid=SID&dsn=DSN'

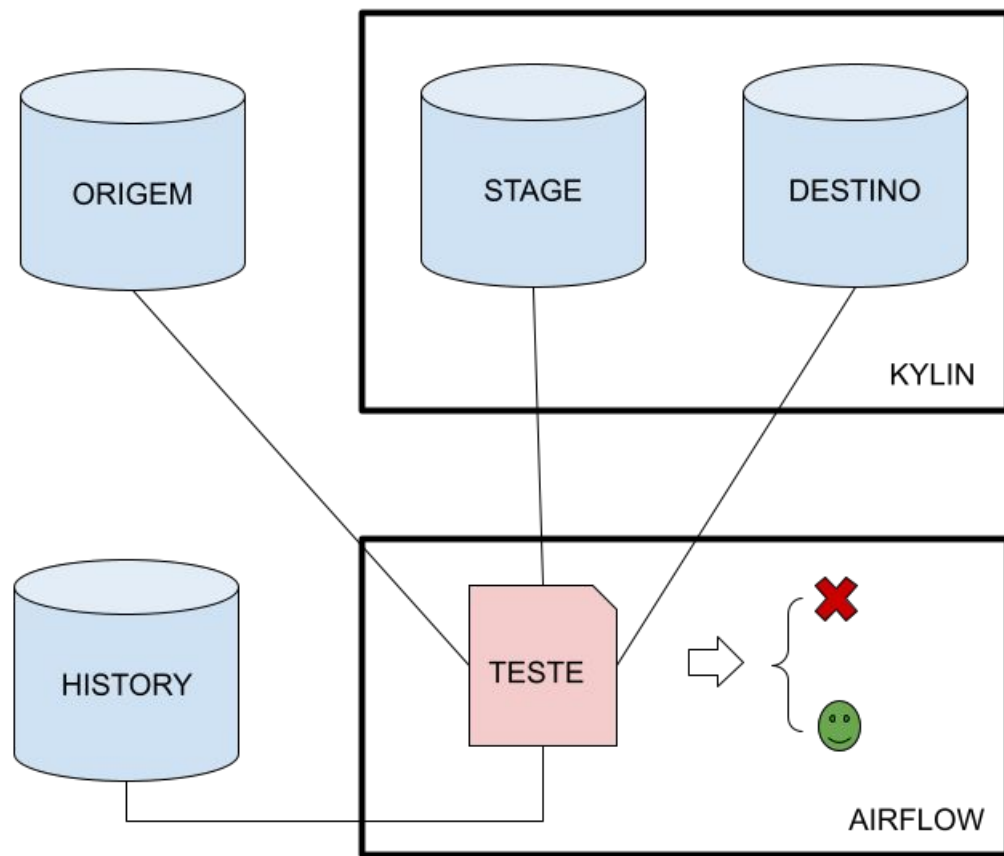
airflow-kylin (jdbc kylin driver)

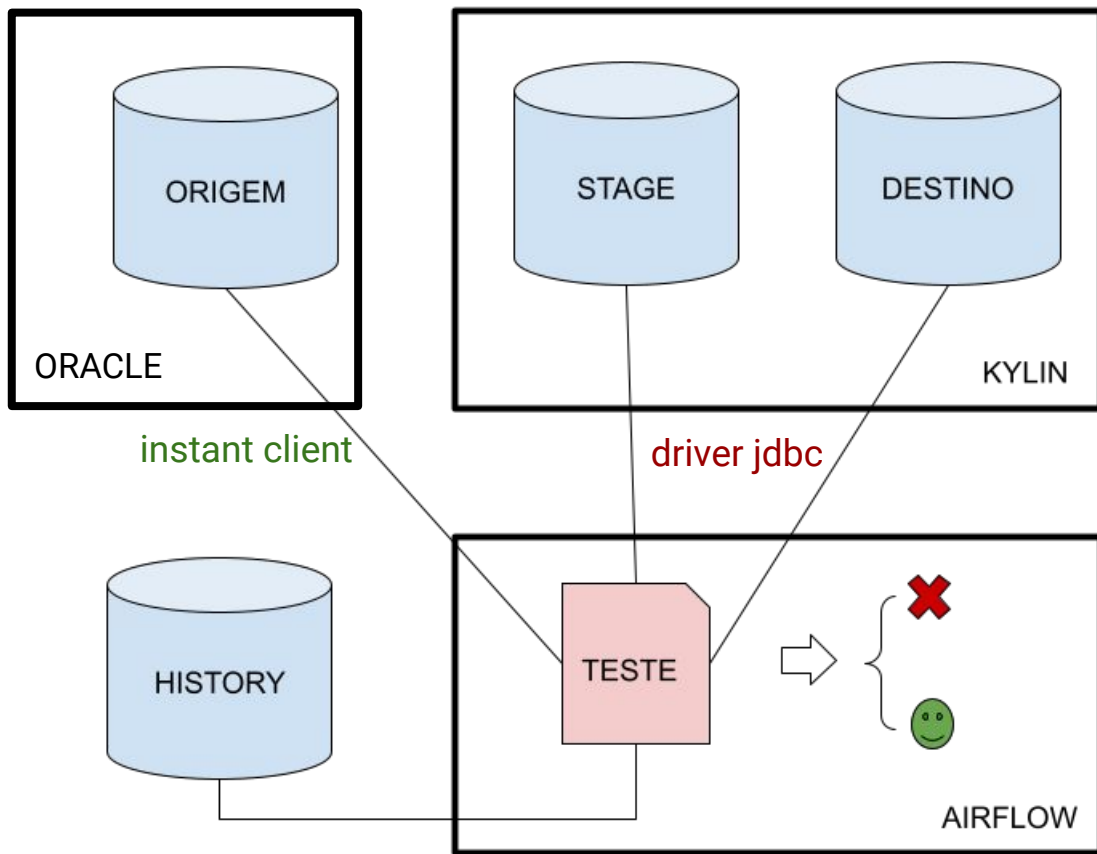


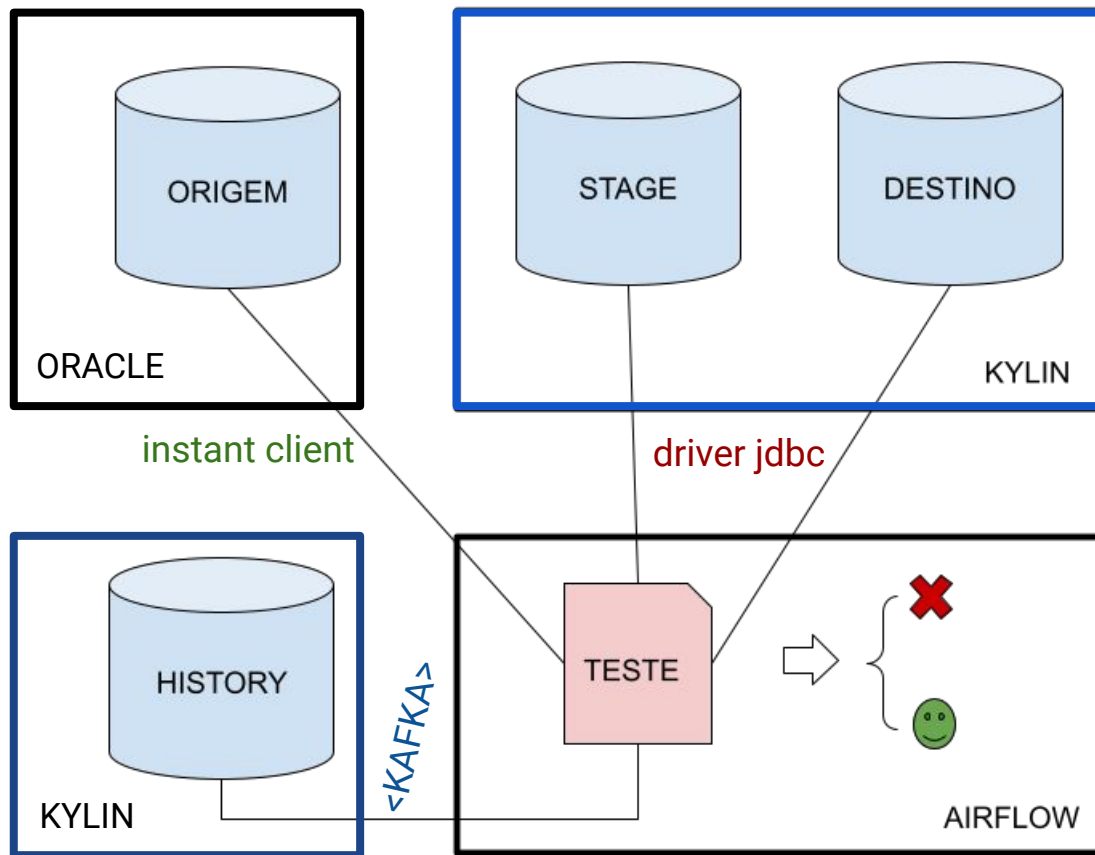


Outras ideias ideais









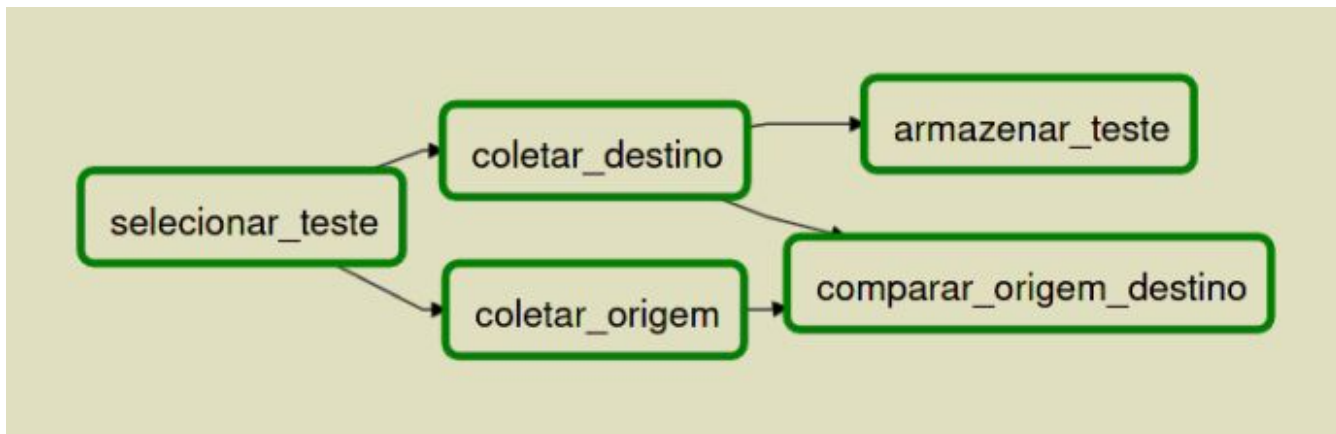
Hands-on (Brains-on)



Hands-on (Brains-on)

- Interface web AIRFLOW
- Apresentação do DAG
 - testeTres.py - *Comparar linhas (agrupadas)*
- Execução manual de um DAG - FALHA (no teste)
 - A) comparação entre tabelas com quantidade de linhas distintas
- Atualização/Sincronização de um DAG
- Execução manual de um DAG - SUCESSO
 - B) comparação entre tabelas com quantidade de linhas iguais
- Execução manual de um DAG - FALHA (no fluxo)
 - C) uma etapa com dependências não é finalizada com sucesso
- Visualização de LOG

Hands-on (Brains-on)



Conceitos

- DAG ----> JOB
- Operator -----> STEP
- Hook -----> CONNECTION

Testes propostos

ID	Teste proposto	Resumo	Tag
1	Série temporal da quantidade de linhas em uma tabela	CONTAR LINHAS	linha
2	Série temporal da quantidade de linhas agrupadas em uma tabela	CONTAR LINHAS AGRUPADAS	grupo
3	Comparação da quantidade de linhas entre tabelas da origem e do destino	COMPARAR LINHAS (AGRUPADAS)	comparacao
4	Validação na origem de colunas que violam restrição de nulidade no destino	VERIFICAR NULOS	nulo

https://docs.google.com/spreadsheets/d/1atubmY6G-Tw4wOHOnZY5mhwtU9WGUH1_rpDCIEF898/edit?usp=sharing

Dá para fazer no PDI? Dá. É de bom tom?

- PDI **ainda** não conecta no big data
- Filosofias diferentes de manutenção do fluxo de trabalho
 - XML *versus* Python
 - ETL *versus* Workflow (de teste)
 - Parametrizações
- AIRFLOW nos liberta com uma linguagem de programação (ao mesmo tempo prende com uma framework)
- O uso do PDI não exclui o Apache AirFlow pois poderemos usar este para executar comandos PDI.
- O uso do PDI facilitaria a "programação" de variados testes que poderia ser orquestrados pelo AirFlow.