



Apache Beam

Introducción, como usarlo y para
que te puede servir

stratebi
open business intelligence



1. INTRODUCCIÓN

Apache Beam es un modelo unificado de código abierto para definir *pipelines* de procesamiento de datos por *lotes* o de *streaming*. Actualmente, Apache Beam SDK tiene soporte para tres lenguajes de programación: Java, Python y Go. Los *pipelines* se programan usando uno de los SDKs mencionados anteriormente y luego se ejecutan en uno de los *back-ends* soportados.

Apache Beam es muy útil a la hora de transformar datos independientes con mucho paralelismo, pero también se puede usar para ETLs e integración de datos.

Los *Runners* traducen el pipeline de procesamiento de datos creado con Beam en una API compatible con el back-end seleccionado. Apache Beam soporta los siguientes back-ends:

- Apache Flink
- Apache Nemo
- Apache Samza
- Apache Spark
- Google Cloud Dataflow
- Hazelcast Jet

En esta primera versión del documento usaremos Python.

2. INSTALACIÓN

Es recomendable usar un entorno virtual para cualquier proyecto en Python para no romper las dependencias del sistema.

Apache Beam SDK requiere Python 2.7 o Python 3.5+. Se puede comprobar la versión con el siguiente mandato:

```
python --version
```

Primero creamos un entorno virtual (*.env* en este caso) en la carpeta del proyecto:

```
python -m venv .env
```

Activamos el entorno virtual:

Unix:

```
. .env/bin/activate
```

Windows (PowerShell):

```
.env\Scripts\activate.ps1
```

Por último, instalamos el paquete *apache-beam* (en este caso con todos los paquetes extra):

```
pip install apache-beam[gcp,aws,test,docs]
```

Podemos comprobar que funciona con el siguiente comando:

```
python -m apache_beam.examples.wordcount --output counts.txt
```

3. PROGRAMACIÓN CON BEAM PYTHON SDK

1. Introducción

Para usar Apache Beam hay que crear un programa *driver* que define un pipeline, con todos las entradas, transformaciones y salidas, más las opciones de ejecución del pipeline.

Apache Beam SDK proporciona las siguientes abstracciones para simplificar la creación de pipelines:

- *Pipeline* – encapsula la tarea de transformación de datos desde el principio hasta el final; todos los programas *driver* tienen que crear un *Pipeline*.
- *PCollection* – un conjunto de datos distribuidos sobre el que opera el *Pipeline*.
- *PTransform* – representa una operación de procesamiento de datos (o un paso) en el *Pipeline*.
- *I/O* – operaciones de entrada / salida.

2. Creación de un Pipeline

El esqueleto de un *Pipeline* es el siguiente:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

with beam.Pipeline(options=PipelineOptions()) as p:
    pass # construir el pipeline aqui
```

Cuando se crea un *Pipeline* también hay que configurar algunas opciones. Estas se pueden configurar programáticamente, pero es mejor configurarlas con anticipación o leerlas desde la línea de comandos y pasarlas al objeto *Pipeline* cuando se crea. Las opciones se usan para configurar diferentes aspectos del *Pipeline*:

- El *Runner* que va a ejecutar el *Pipeline*
- Las opciones del *Runner*
- El ID del proyecto

- La ruta de los ficheros
- Etc.

Beam SDK incluye un parser de línea de comandos para poder pasar las opciones en el momento de la ejecución del *Pipeline*. *PipelineOptions* se puede extender fácilmente para añadir opciones personalizadas.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

class MisOpciones(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
        parser.add_argument('--input')
        parser.add_argument('--output')

with beam.Pipeline(options=MisOpciones()) as p:
    pass  # construir el pipeline aqui
```

La función *parser.add_argument* admite varios parámetros entre cuales los más usados son *help* para especificar una descripción de la opción, y *default* para definir un valor por defecto.

3. PCollections

Las *PCollections* se pueden crear leyendo datos de fuentes externas o definir los datos en memoria.

Ejemplo de lectura de un fichero:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

with beam.Pipeline(options=PipelineOptions()) as p:
    lineas = p | 'LeerDatos' >> beam.io.ReadFromText('./input/data.txt') \
        | beam.Map(print)
```

Ejemplo de creación de una *PCollection* con datos en memoria:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

with beam.Pipeline(options=PipelineOptions()) as p:
    lineas = p | beam.Create(['test']) \
        | beam.Map(print)
```

Algunas características importantes de las *PCollections* son:

- Una *PCollection* es propiedad de un *Pipeline* específico y no se puede compartir.
- Los elementos pueden ser de cualquier tipo, pero todos deben ser del mismo tipo.
- Pueden tener esquemas.

- Son inmutables, las transformaciones pueden procesar los elementos y generar nuevos datos, pero no consumen ni modifican la colección original.
- No admiten el acceso aleatorio a elementos individuales.
- Una *PCollection* puede tener un tamaño limitado o ilimitado; la lectura de un fichero crea una *PCollection* con tamaño limitado, el streaming (p.e. Kafka) crea una *PCollection* con tamaño ilimitado.
- Cada elemento de una *PCollection* tiene una marca de tiempo intrínseca asociada que se asigna cuando se crea la *PCollection*; en el caso de datos streaming, muchas veces, la marca de tiempo corresponde al momento en que se ha leído el elemento.

4. Transformaciones

Las transformaciones son las operaciones del *Pipeline*. Se aplican a cada elemento de una *PCollection* de entrada. Para invocar una transformación hay que usar el operador de tubería. Las transformaciones se pueden encadenar y tienen la siguiente estructura:

```
Output = Input | PrimeraTransformacion | SegundaTransformacion | ...
```

Beam proporciona varias transformaciones principales.

a. ParDo

Transformación para procesamiento en paralelo, realiza alguna función de procesamiento para cada elemento de una *PCollection* de entrada y emite cero, uno o varios elementos en una *PCollection* de salida. Al aplicar una transformación *ParDo* se tiene que proporcionar un objeto derivado de *DoFn*. La implementación de la función *process* contiene toda la lógica de procesamiento y debe devolver un iterable con todos los valores de salida.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

class LongitudPalabraFn(beam.DoFn):
    def process(self, element):
        return [len(element)]

palabras = ['Hola', 'Mundo']

with beam.Pipeline(options=PipelineOptions()) as p:
    longs = p | beam.Create(palabras) \
              | beam.ParDo(lambda palabra: [len(palabra)]) \
              | beam.Map(print)
```

Resultado:

4
5

ParDo también acepta funciones lambda.

b. GroupByKey

Transformación para procesar colecciones de pares clave / valor. Produce una colección donde cada elemento consiste en una clave y todos los valores asociados a esa clave.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

mascotas = [('gato', 'Ragdoll'),
             ('gato', 'Ocigato'),
             ('perro', 'Beagle'),
             ('perro', 'Golden Retriever')]

with beam.Pipeline(options=PipelineOptions()) as p:
    agrupados = p | beam.Create(mascotas) \
                  | beam.GroupByKey() \
                  | beam.Map(print)
```

Resultado:

```
('gato', ['Ragdoll', 'Ocigato'])
('perro', ['Beagle', 'Golden Retriever'])
```

c. CoGroupByKey

Transformación para realizar la unión relacional entre dos o más colecciones de pares clave / valor.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

with beam.Pipeline(options=PipelineOptions()) as p:
    tipos_mascotas = p | 'Crear tipos' \
                      >> beam.Create([('gato', 'Ragdoll'),
                                       ('gato', 'Ocigato'),
                                       ('perro', 'Beagle'),
                                       ('perro', 'Golden Retriever')])

    esperanza_vida = p | 'Crear esperanza de vida' \
                      >> beam.Create([('gato', 15),
                                       ('perro', 12)])

    mascotas = ({'tipos': tipos_mascotas, 'esperanza_vida': esperanza_vida}
                  | beam.CoGroupByKey()
                  | beam.Map(print))
```

Resultado:

```
( 'gato', { 'tipos': [ 'Ragdoll', 'Ocigato' ], 'esperanza_vida': [15] })
( 'perro', { 'tipos': [ 'Beagle', 'Golden Retriever' ], 'esperanza_vida': [12] })
```

d. Combine

Grupo de transformaciones para combinar colecciones de elementos o valores. Las transformaciones de combinación simples se pueden implementar con funciones.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

def mi_suma(valores):
    return sum(valores)

with beam.Pipeline(options=PipelineOptions()) as p:
    suma = p | 'Crear lista numeros' >> beam.Create([1, 2, 3, 4]) \
            | beam.CombineGlobally(lambda valores: sum(valores)) \
            | 'Suma' >> beam.Map(print)

    suma_claves = p | 'Crear pares' >> beam.Create([ ('impar', 1),
                                                    ('par', 2),
                                                    ('impar', 3),
                                                    ('par', 4) ]) \
                | beam.CombinePerKey(sum) \
                | 'Suma por claves' >> beam.Map(print)

    suma_valores = p | 'Crear valores' >> beam.Create([ ('impar', [3, 5]),
                                                        ('par', [4, 6]) ]) \
                | beam.CombineValues(mi_suma) \
                | 'Suma valores' >> beam.Map(print)
```

Resultado:

```
10
('impar', 4)
('par', 6)
('impar', 8)
('par', 10)
```

Las transformaciones de combinación más complejas requieren crear subclases de *CombineFn*. Para más información: [CombineGlobally](#).

e. Flatten

Fusiona varias *PCollections* en una única *PCollection* lógica.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

perros = [ 'Beagle', 'Golden Retriever' ]
gatos = [ 'Ocigato', 'Ragdoll' ]

with beam.Pipeline(options=PipelineOptions()) as p:
```

```
perros = p | 'Crear perros' >> beam.Create(['Beagle', 'Golden Retriever'])
gatos = p | 'Crear gatos' >> beam.Create(['Ocigato', 'Ragdoll'])
mascotas = (perros, gatos) | beam.Flatten() | beam.Map(print)
```

Resultado:

```
Ocigato
Ragdoll
Beagle
Golden Retriever
```

f. Partition

Divide una *PCollection* en un número fijo de colecciones más pequeñas.

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

especies = ['gato', 'perro']

def por_especie(mascota, num_particiones):
    return especies.index(mascota[0])

with beam.Pipeline(options=PipelineOptions()) as p:
    gatos, perros = p | beam.Create([('gato', 'Ragdoll'),
                                      ('perro', 'Beagle'),
                                      ('gato', 'Ocigato'),
                                      ('perro', 'Golden Retriever')]) \
        | beam.Partition(por_especie, len(especies))

    gatos | 'Gatos' >> beam.Map(lambda elem: print(f'Part0: {elem}'))
    perros | 'Perros' >> beam.Map(lambda elem: print(f'Part1: {elem}'))
```

Resultado:

```
Part0: ('gato', 'Ragdoll')
Part1: ('perro', 'Beagle')
Part0: ('gato', 'Ocigato')
Part1: ('perro', 'Golden Retriever')
```

g. Otras

Beam Python SDK proporciona más transformaciones. Ver [Transform catalog](#).

5. Entrada / Salida (I/O)

Beam proporciona varios componentes para las operaciones de entrada / salida. El Python SDK tiene menos componentes que el Java SDK. Para más información: [I/O](#).

6. Ventanas

Algunas transformaciones agrupan varios elementos por una clave común. Con un conjunto de datos ilimitado (streaming), es imposible recopilar todos los elementos, ya que se leen constantemente nuevos elementos. Si está trabajando con *PCollections* ilimitadas, las ventanas son especialmente útiles.

Las ventanas se crean con la siguiente clase:

```
beam.WindowInto(<window>)
```

<window> puede ser:

- `FixedWindows(size)` – representa un intervalo de tiempo de duración constante, sin superposición en el flujo de datos.
- `SlidingWindows(size, period)` – representa intervalos de tiempo en el flujo de datos, pueden superponerse (si `period < size`).
- `Sessions(gap_size)` – forma una nueva ventana cuando ha pasado un intervalo de tiempo mínimo entre un elemento y el siguiente.
- `GlobalWindows` – ventana por defecto.

7. Triggers

Al recopilar y agrupar datos en ventanas, Beam usa disparadores para determinar cuándo emitir los resultados agregados de cada ventana. Por defecto genera los datos cuando estima que han llegado todos los datos y descarta los datos posteriores para esa ventana.

Beam proporciona una serie de triggers prediseñados:

- *AfterWatermark* – opera en el tiempo del evento; emite el contenido de una ventana después de que el *watermark* pasa el final de la ventana, según las marcas de tiempo adjuntas a los elementos de datos; *watermark* es una métrica de progreso global y representa la completitud de las entradas dentro de su *Pipeline* en cualquier momento.
- *AfterProcessingTime* – opera en el tiempo de procesamiento; emite una ventana después de que ha pasado una cierta cantidad de tiempo de procesamiento desde que se recibieron los datos; el tiempo de procesamiento lo determina el reloj del sistema, en lugar de la marca de tiempo del elemento de datos.
- *AfterCount* – se activa después de que se hayan recopilado al menos N elementos.

Cuando se usan triggers, también se debe establecer el modo de acumulación de la ventana. El modo de acumulación determina si el sistema acumula o descarta.

Si se desea que el *Pipeline* procese los datos que llegan después de que el *watermark* pase el final de la ventana, se puede especificar *allowed_lateness* en la configuración de la ventana.

Beam permite combinar varios *triggers* para formar *triggers compuestos*. También permite especificar si los resultados se emiten repetidamente, como máximo una vez o en otras condiciones personalizadas.

Beam proporciona una serie de *triggers* compuestos:

- *Repeatedly* - se ejecuta repetidamente.
- *AfterEach* - combina varios *triggers* para disparar en una secuencia específica.
- *AfterFirst* - equivalente a una operación lógica OR para múltiples *triggers*.
- *AfterAll* - equivalente a una operación lógica Y para múltiples *triggers*.

```
p | WindowInto(  
    FixedWindows(60),  
    trigger=Repeatedly(  
        AfterAny(AfterCount(100), AfterProcessingTime(60))),  
    accumulation_mode=AccumulationMode.DISCARDING)
```

4. RUNNERS

Un *Runner* ejecuta un *Pipeline* de Beam en un sistema de procesamiento de datos específico. Veremos los *Runners* soportados por Beam Python SDK.

1. Direct Runner

Se usa para desarrollo y pruebas de *Pipelines*. Para especificar el paralelismo basta con ejecutar el *Pipeline* con la opción `--direct_num_workers`. El valor 0 establece el paralelismo en el número de núcleos de la máquina donde se está ejecutando el *Pipeline*. Para más información: [Direct Runner](#).

2. Apache Flink

Para usar Apache Flink hay que ejecutar el *Pipeline* con `--runner=FlinkRunner`. Con la opción `--flink_master` se especifica la conexión con el nodo maestro del clúster. Para más información: [Apache Flink](#).

3. Apache Spark

Para usar Apache Spark hay que ejecutar el *Pipeline* con `--runner=SparkRunner`. Con la opción `--spark_master_url` se especifica la conexión con el nodo maestro del clúster. Para más información: [Apache Spark](#).

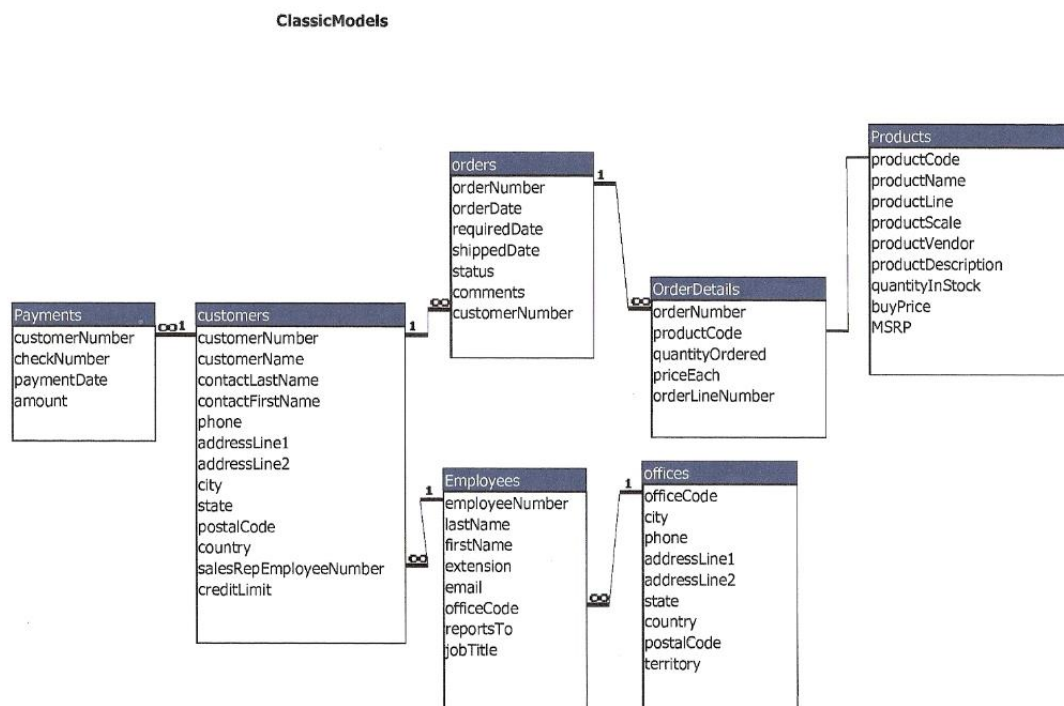
4. Google Cloud Dataflow

Google Cloud Dataflow es un servicio para ejecutar *Pipelines* de Apache Beam dentro del ecosistema de Google Cloud Platform. Para más información: [Google Cloud Dataflow Runner](#), [Google Cloud Dataflow](#).

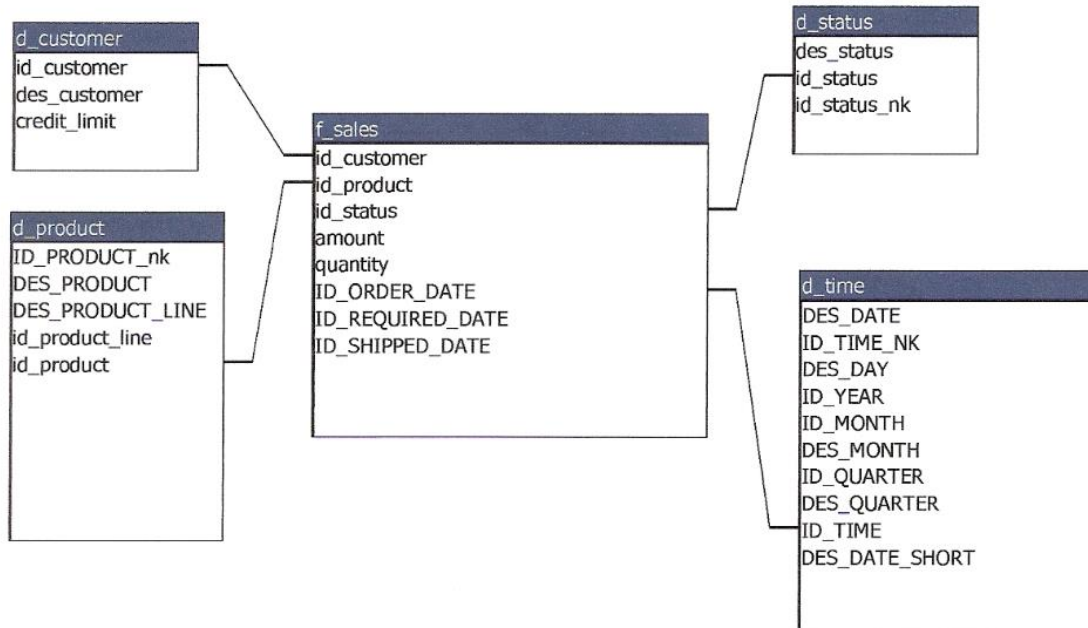
5. EJEMPLO ETL

Partiendo de una base de datos se va a implementar un proceso ETL para generar y cargar un Datamart de Ventas.

A continuación, se muestra el diagrama Entidad / Relación del origen:



El diagrama del Datamart destino es el siguiente:



En esta versión del documento no se crea la dimensión `d_time`. Tenga en cuenta que las transformaciones utilizadas no son necesariamente las óptimas, ya que en este ejemplo queremos enseñar las diferentes funcionalidades de Apache Beam.

Dado que Python SDK no tiene soporte para lectura / escritura de bases de datos, hay que instalar un paquete adicional:

```
pip install beam-nuggets
```

a. Código inicial

Usamos el esqueleto del pipeline definido anteriormente, importamos los paquetes necesarios (`sqlalchemy` para definir los metadatos de las tablas y `beam_nuggets` para conectar con bases de datos relacionales) añadimos opciones personalizadas para la conexión con la base de datos:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
from beam_nuggets.io import relational_db
from beam_nuggets import transforms
from sqlalchemy import Table, Column, INT, VARCHAR, BIGINT

class DatabaseOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
        super()._add_argparse_args(parser)
        parser.add_argument('--db_driver', default='mysql+pymysql')
        parser.add_argument('--db_host', default='localhost')
        parser.add_argument('--db_port', default=3306, type=int)
        parser.add_argument('--db_username', default='root')
```

```

        parser.add_argument('--db_password', default='secret_password')
        parser.add_argument('--db_name', default='sampledata_etl')

pipeline_options = DatabaseOptions()

source_config = relational_db.SourceConfiguration(
    drivename=pipeline_options.db_driver,
    host=pipeline_options.db_host,
    port=pipeline_options.db_port,
    username=pipeline_options.db_username,
    password=pipeline_options.db_password,
    database=pipeline_options.db_name,
)

with beam.Pipeline(options=pipeline_options) as p:
    pass # construir Pipeline aquí

```

b. Dimensión d_customer

Partimos de la tabla *customer*, seleccionamos las columnas *CUSTOMERNUMBER*, *CUSTOMERNAME*, *CREDITLIMIT*. Luego renombramos las columnas *CUSTOMERNUMBER* a *ID_CUSTOMER* y *CUSTOMERNAME* a *DES_CUSTOMER*. Guardamos el resultado en la tabla *d_customer*.

```

d_customer = p \
    | 'Leer customers' >>
relational_db.ReadFromDB(source_config=source_config,

table_name='customers',

CUSTOMERNUMBER, '                                query='SELECT
                                                'CUSTOMERNAME, '
                                                'CREDITLIMIT '
                                                'FROM

customers') \
    | 'Renombrar columnas customer' >> beam.ParDo(RenombrarColumnasFn(),

col_map={'CUSTOMERNUMBER': 'ID_CUSTOMER',

'CUSTOMERNAME': 'DES_CUSTOMER'})

d_customer | 'Escribir en d_customer' >>
relational_db.Write(source_config=source_config,

table_config=d_customer_table)

```

La clase usada para el renombrado es la siguiente:

```

class RenombrarColumnasFn(beam.DoFn):
    def process(self, element, *args, **kwargs):
        if 'col_map' in kwargs:
            for k, v in kwargs['col_map'].items():
                element[v] = element.pop(k)
            return [element]

```

La función *process* de esta clase admite un parámetro *col_map* de tipo diccionario con el mapeo de las columnas a renombrar.

La configuración de la tabla *d_customer*es:

```
def d_customer_ddl(metadata):
    return Table(
        'd_customer', metadata,
        Column('ID_CUSTOMER', INT, primary_key=True),
        Column('DES_CUSTOMER', VARCHAR(50)),
        Column('CREDITLIMIT', BIGINT)
    )

d_customer_table = relational_db.TableConfiguration(
    name='d_customer',
    define_table_f=d_customer_ddl,
    create_if_missing=True,
)
```

c. Dimensión *d_status*

Seleccionamos la columna *STATUS* de la tabla *orders*, convertimos el diccionario en tuplas y seleccionamos los valores distintos. Luego convertimos de nuevo a diccionario, generamos una secuencia con el nombre *ID_STATUS*, inicializamos *ID_STATUS_NK* a *STATUS* y cambiamos el nombre de la columna *STATUS* a *DES_STATUS*. Por último, guardamos los datos en la tabla *d_status*.

```
d_status = p \
    | 'Leer status' >>
relational_db.ReadFromDB(source_config=source_config,
                          table_name='orders',
                          query='SELECT STATUS FROM
orders') \
    | 'Tuplas status' >> beam.Map(lambda e: list(e.items())[0]) \
    | 'Tuplas distintas' >> beam.Distinct() \
    | 'Tuplas a dict' >> beam.Map(lambda e: dict([e])) \
    | 'Generar secuencia' >>
beam.ParDo(transforms.AssignUniqueId('ID_STATUS')) \
    | 'Generar ID_STATUS_NK' >> beam.ParDo(lambda e: [{**e,
'ID_STATUS_NK': e['STATUS']}]) \
    | 'Renombrar columnas status' >> beam.ParDo(RenombrarColumnasFn(),
col_map={'STATUS':
'DES_STATUS'})

d_status | 'Escribir en d_status' >>
relational_db.Write(source_config=source_config,
table_config=d_status_table)
```

La configuración de la tabla *d_status* es la siguiente:

```
def d_status_ddl(metadata):
    return Table(
        'd_status', metadata,
```

```

        Column('ID_STATUS', INT, primary_key=True),
        Column('DES_STATUS', VARCHAR(15)),
        Column('ID_STATUS_NK', VARCHAR(15))
    )

d_status_table = relational_db.TableConfiguration(
    name='d_status',
    define_table_f=d_status_ddl,
    create_if_missing=True
)

```

d. Dimensión d_product

En el primer paso seleccionamos las columnas necesarias de la tabla *products* y generamos *ID_PRODUCT*.

En el segundo paso seleccionamos los valores distintos de la columna *PRODUCTLINE* y generamos *ID_PRODUCT_LINE*.

En el tercer paso hacemos un left join entre la colección *products* y *product_lines*, y cambiamos los nombres de *PRODUCTCODE* a *ID_PRODUCT_NK*, *PRODUCTNAME* a *DES_PRODUCT*, *PRODUCTLINE* a *DES_PRODUCT_LINE*.

Por último, escribimos los datos en la tabla *d_product*

```

products = p \
    | 'Leer product' >>
relational_db.ReadFromDB(source_config=source_config,
                           table_name='products',
                           query='SELECT
PRODUCTCODE, '
                                'PRODUCTNAME, '
                                'PRODUCTLINE '
                                'FROM products') \
    | 'Generar ID_PRODUCT' >>
beam.ParDo(transforms.AssignUniqueId('ID_PRODUCT'))

product_lines = p \
    | 'Leer product line' >>
relational_db.ReadFromDB(source_config=source_config,
                           table_name='products',
                           query='SELECT '
                                'DISTINCT
PRODUCTLINE '
                                'FROM
products') \
    | 'Secuencia product line' >>
beam.ParDo(transforms.AssignUniqueId('ID_PRODUCT_LINE'))

d_product = products \
    | 'products Left Join product_lines' >> LeftJoin(product_lines,
    'PRODUCTLINE', 'PRODUCTLINE') \

```



```

    | 'Renombrar columnas d_product' >> beam.ParDo(RenombrarColumnasFn(),
col_map={'PRODUCTCODE': 'ID_PRODUCT_NK',
'PRODUCTNAME': 'DES_PRODUCT',
'PRODUCTLINE': 'DES_PRODUCT_LINE'})

d_product | 'Escribir en d_product' >>
relational_db.Write(source_config=source_config,
table_config=d_product_table)

```

LeftJoin es una transformación compuesta y se ha definido como:

```

class LeftJoin(beam.PTransform):
    class LeftJoinFn(beam.DoFn):
        def process(self, element, *args, **kwargs):
            for item in element['main']:
                try:
                    item = {**item, **element['lookup'][0]}
                    yield item
                except IndexError:
                    yield item

    def __init__(self, lookup, left_on, right_on, label=None):
        super().__init__(label)
        self.lookup = lookup
        self.left_on = left_on
        self.right_on = right_on

    def expand(self, main):
        cc_main = main \
            | f'{self.label}: clave comun main' >> beam.Map(lambda e:
(e[self.left_on], e))
        cc_lookup = self.lookup \
            | f'{self.label}: clave comun lookup' >> beam.Map(lambda e:
(e[self.right_on], e))
        return (
            {'main': cc_main, 'lookup': cc_lookup}
            | f'{self.label}: CoGroupByKey main y lookup' >>
beam.CoGroupByKey()
            | f'{self.label}: borrar clave comun' >> beam.Map(lambda elem:
elem[1])
            | f'{self.label}: Left Join' >> beam.ParDo(self.LeftJoinFn())
)

```

La transformación admite tres parámetros en su constructor:

- `lookup` – es la colección de búsqueda.
- `left_on` – es la clave de la colección a la izquierda del join.
- `right_on` – es la clave de la colección a la derecha del join.

La configuración de la tabla *d_product* es la siguiente:

```
def d_product_ddl(metadata):
    return Table(
        'd_product', metadata,
        Column('ID_PRODUCT', BIGINT, primary_key=True),
        Column('ID_PRODUCT_NK', VARCHAR(50)),
        Column('DES_PRODUCT', VARCHAR(70)),
        Column('ID_PRODUCT_LINE', INT),
        Column('DES_PRODUCT_LINE', VARCHAR(50))
    )

d_product_table = relational_db.TableConfiguration(
    name='d_product',
    define_table_f=d_product_ddl,
    create_if_missing=True
)
```

e. Tabla de hechos *f_sales*

En el primer paso obtenemos las columnas necesarias de la tabla *orders*.

En el segundo paso obtenemos las columnas necesarias de la tabla *orderdetails*, hacemos left join con la colección *orders* para recuperar los campos *STATUS* y *CUSTOMERNUMBER*, creamos el campo *AMOUNT* multiplicando *PRICEEACH* y *QUANTITYORDERED*. Luego hacemos dos left joins, uno con *d_status* para obtener el *ID_STATUS*, y otro con *d_product* para recuperar *ID_PRODUCT*. Seleccionamos las columnas que nos interesa, cambiamos los nombres de *CUSTOMERNUMBER* a *ID_CUSTOMER* y *QUANTITYORDERED* a *QUANTITY*. Seleccionamos las claves para luego agrupar por estas claves. Esta transformación devuelve una lista de diccionarios por clave. Obtenemos los valores (las listas) descartando las claves, y transformamos las lista en diccionario cogiendo los campos que no se agregan y agregando los campos *AMOUNT* y *QUANTITY* con una función suma.

Por último, guardamos los datos en la tabla *f_sales*.

```

orders = p \
    | 'Leer orders' >> relational_db.ReadFromDB(source_config=source_config,
                                                table_name='orders',
                                                query='SELECT ORDERNUMBER, '
                                                    'STATUS, '
                                                    'CUSTOMERNUMBER '
                                                    'FROM orders')

f_sales = p \
    | 'Leer orderdetails' >>
relational_db.ReadFromDB(source_config=source_config,
table_name='orderdetails',
                                                                    query='SELECT
ORDERNUMBER, '
                                                                    'PRODUCTCODE, '
'QUANTITYORDERED, '
                                                                    'PRICEEACH, '
'ORDERLINENUMBER '
                                                                    'FROM
orderdetails') \
    | 'Recuperar orders' >> LeftJoin(orders, 'ORDERNUMBER', 'ORDERNUMBER')
\
    | 'Calcular AMOUNT' >> beam.Map(lambda e: {**e, 'AMOUNT':
e['PRICEEACH'] * e['QUANTITYORDERED']}) \
    | 'Recuperar ID_STATUS' >> LeftJoin(d_status, 'STATUS', 'ID_STATUS_NK')
\
    | 'Recuperar ID_PRODUCT' >> LeftJoin(d_product, 'PRODUCTCODE',
'ID_PRODUCT_NK') \
    | 'Seleccionar columnas' >> beam.ParDo(SeleccionarColumnasFn(),
                                                columns=['CUSTOMERNUMBER',
                                                        'ID_PRODUCT',
                                                        'ID_STATUS',
                                                        'AMOUNT',
                                                        'QUANTITYORDERED']) \
    | 'Renombrar columnas' >> beam.ParDo(RenombrarColumnasFn(),
                                                col_map={'CUSTOMERNUMBER':
'ID_CUSTOMER',
                                                        'QUANTITYORDERED':
'QUANTITY'}) \
    | 'Claves para agrupar' >> beam.Map(lambda e: ((e['ID_CUSTOMER'],
e['ID_PRODUCT'],
e['ID_STATUS']),
e)) \
    | 'CombineByKey' >> beam.GroupByKey() \
    | 'Valores' >> beam.Values() \
    | 'Sumar grupo' >> beam.Map(lambda e: {**e[0], **sumar_columnas(e,
['AMOUNT', 'QUANTITY'])})

f_sales | relational_db.Write(source_config=source_config,
                             table_config=f_sales_table)
    
```

La clase para seleccionar columnas es la siguiente:

```
class SeleccionarColumnasFn(beam.DoFn):
    def process(self, element, *args, **kwargs):
        if 'columnas' in kwargs:
            resultado = {}
            for columna in kwargs['columnas']:
                resultado[columna] = element[columna]
            return [resultado]
```

La función *process* de esta clase admite el argumento *columnas* de tipo lista y sirve para especificar las columnas a seleccionar.

En la agregación se ha usado una función llamada *sumar_columnas*. La definición de esta función es la siguiente:

```
def sumar_columnas(lista, columnas):
    resultado = {}
    for columna in columnas:
        resultado[columna] = 0

    for elem in lista:
        for columna in columnas:
            resultado[columna] += elem[columna]

    return resultado
```

La función admite dos argumentos, la *lista* de diccionarios y las *columnas* (los campos) para los que se aplica la suma.

La configuración de la tabla *f_sales* es la siguiente:

```
def f_sales_ddl(metadata):
    return Table(
        'f_sales', metadata,
        Column('num_fila', BIGINT, primary_key=True),
        Column('ID_CUSTOMER', INT),
        Column('ID_PRODUCT', INT),
        Column('ID_STATUS', INT),
        Column('QUANTITY', INT),
        Column('AMOUNT', INT)
    )

f_sales_table = relational_db.TableConfiguration(
    name='f_sales',
    define_table_f=f_sales_ddl,
    create_if_missing=True
)
```

f. Integración con Apache Airflow

Usaremos Apache Airflow para orquestar el proceso ETL (el Pipeline) definido anteriormente con Apache Beam.

Para esto tenemos que instalar el paquete *apache-airflow*:

```
pip install apache-airflow
```

Creamos un nuevo fichero py con el siguiente código:

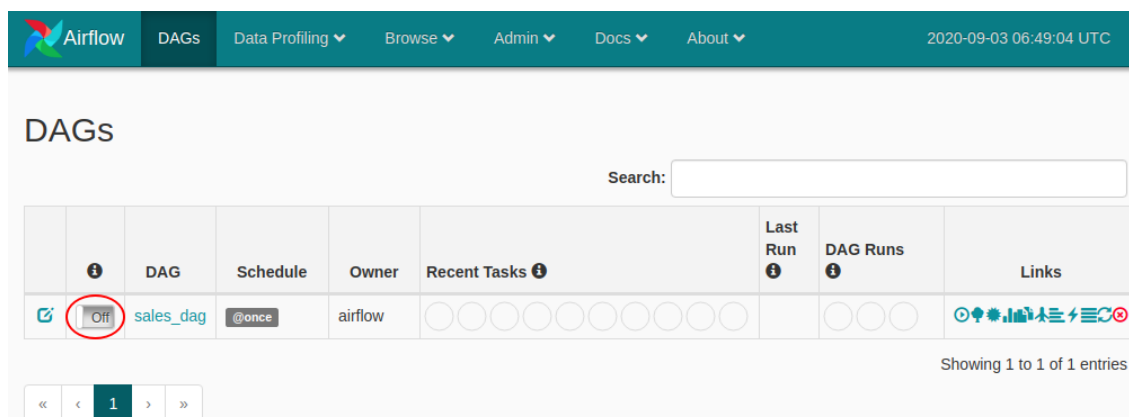
```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

dag = DAG(
    dag_id='sales_dag',
    start_date=days_ago(0),
    schedule_interval='@once'
)

sales_etl = BashOperator(
    task_id='sales_etl',
    bash_command='python /ETLs/sales_etl.py --db_host=mysql ',
    dag=dag
)
```

En el código definimos un DAG con id *sales_dag*, fecha de inicio hoy (*days_ago(0)*) y que se ejecute solo una vez (*@once*). Creamos una tarea usando el operador *BashOperator* y le pasamos el comando para ejecutar el Pipeline. Copiamos el fichero en la carpeta dags de una instancia de Apache Airflow (local o en la nube) y navegamos a la interfaz de este.

Activamos el DAG pinchando en la casilla *Off*.



The screenshot shows the Apache Airflow web interface. At the top, there's a navigation bar with 'Airflow' and 'DAGs' tabs. Below the navigation bar, the 'DAGs' section is active. It features a search bar and a table of DAGs. The table has columns for 'DAG', 'Schedule', 'Owner', 'Recent Tasks', 'Last Run', 'DAG Runs', and 'Links'. The first row shows a DAG named 'sales_dag' with a schedule of '@once' and owner 'airflow'. The 'Off' button next to the DAG name is circled in red. Below the table, it says 'Showing 1 to 1 of 1 entries'.

El DAG se va a ejecutar una vez.

Airflow DAGs							
2020-09-03 09:00							
DAGs							
Search: <input type="text"/>							
		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs
	On	sales_dag	@once	airflow	<div><div>1</div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>	2020-09-03 00:00	<div><div>1</div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>

6. ¿POR QUÉ APACHE BEAM?

- Portabilidad entre los motores de procesamiento de datos.
- API unificado para el procesamiento batch y streaming.
- Integrado con el ecosistema de Google Cloud.

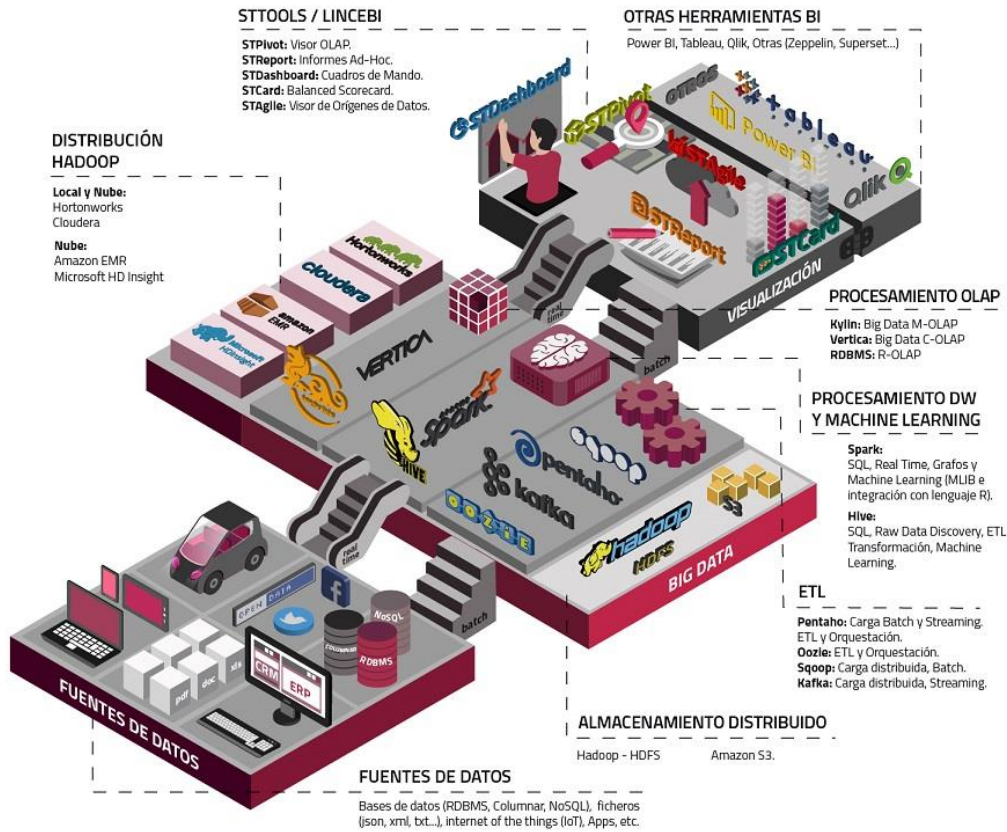
7. CONCLUSIONES

Apache Beam es un framework que permite crear Pipelines con solo una base de código para varios back-ends (Apache Spark, Apache Flink, Google Cloud Dataflow...). Soporta tres lenguajes de programación: Java, Python y Go, pero el SDK de Java tiene más funciones que las otras dos. En esta versión hemos usado Python para explorar las funciones de Apache Beam, en una versión futura utilizaremos Java para ver las características más complejas del framework.

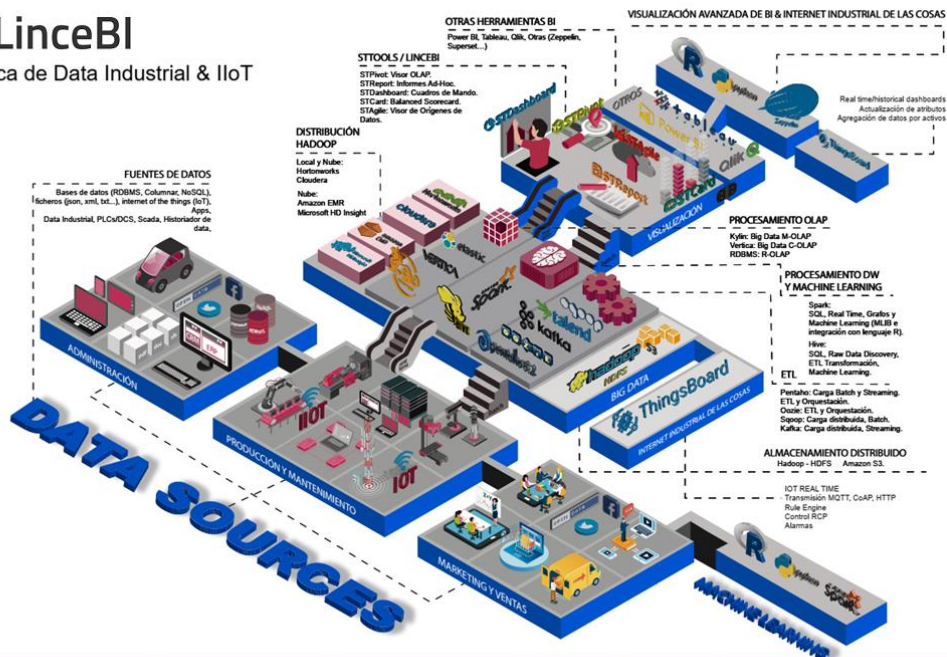
8. TECNOLOGÍAS

Recientemente, hemos sido nombrados Partners Certificados de Vertica, Talend, Microsoft, Snowflake, Kylligence, Pentaho, etc.





LinceBI
Análítica de Data Industrial & IIoT



INTELIGENCIA ARTIFICIAL	INTERNET INDUSTRIAL DE LAS COSAS	ROBÓTICA AUTOMATIZADA DE PROCESOS	BIG DATA	MACHINE LEARNING	BUSINESS INTELLIGENCE
-------------------------	----------------------------------	-----------------------------------	----------	------------------	-----------------------

9. INFORMACIÓN SOBRE STRATEBI



Stratebi es una empresa española, con sede en Madrid y oficinas en Barcelona, Alicante y Sevilla, con amplia experiencia en sistemas de información, soluciones tecnológicas y procesos relacionados con soluciones de Open Source y de inteligencia de Negocio.

Esta experiencia, adquirida durante la participación en proyectos estratégicos en compañías de reconocido prestigio a nivel internacional, se ha puesto a disposición de nuestros clientes.

Somos **Partners Certificados en Microsoft PowerBI** con una dilatada experiencia

Stratebi es la única empresa española que ha estado presente todos los Pentaho Developers celebrados en Europa habiendo organizado el de España.

En Stratebi nos planteamos como **objetivo** dotar a las compañías e instituciones, de herramientas escalables y adaptadas a sus necesidades, que conformen una estrategia Business Intelligence capaz de rentabilizar la información disponible. Para ello, nos basamos en el desarrollo de soluciones de Inteligencia de Negocio, mediante tecnología Open Source.

Stratebi son **profesores y responsables de proyectos** del Master en Business Intelligence de la Universidad UOC, UCAM, EOI...

Los profesionales de Stratebi son los creadores y autores del primer weblog en español sobre el mundo del Business Intelligence, Data Warehouse, CRM, Dashboards, Scorecard y Open Source. Todobi.com

Stratebi es partner de las principales soluciones Analytics: Microsoft Power BI, Talend, Pentaho, Vertica, Snowflake, Kylogence, Cloudera...

Todo Bi, se ha convertido en una referencia para el conocimiento y divulgación del Business Intelligence en español.

10. OTROS

Trabajamos en los principales sectores y con algunas de las compañías y organizaciones más importantes de España.

SECTOR PRIVADO



SECTOR PÚBLICO



11. EJEMPLOS DE DESARROLLOS ANALYTICS

A continuación, se presentan **ejemplos de algunos screenshots** de cuadros de mando diseñados por Stratebi, con el fin de dar a conocer lo que se puede llegar a obtener, así como Demos Online en la web de Stratebi:

