TOPIC: DATA PROVENANCE

Goal: support to explanation of results of big data processing (big data debugger).

Useful for unexpected results/odd performance.

Examples: Scorpion (https://github.com/sirrice/scorpion), Titian.

Technique: data provenance.

From the results of a pipeline of operations, get:

the origin of results

who modified the data and how

Problem at study: data provenance data-processing in Machine Learning.

PROVENIENZA

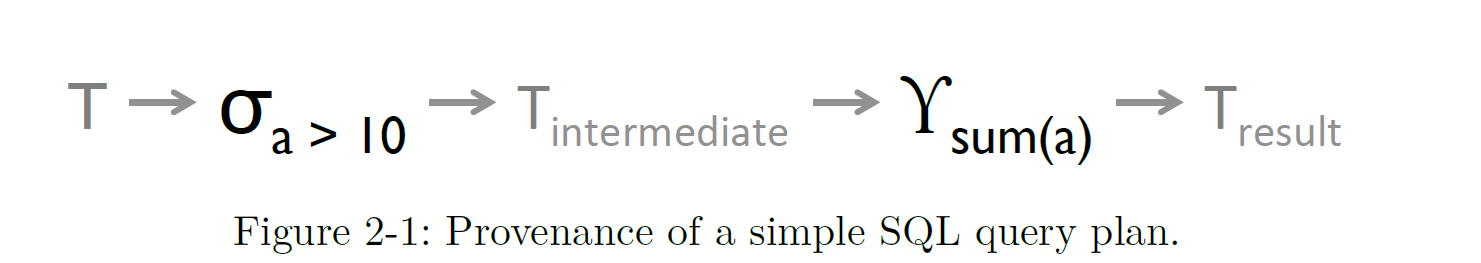
Il termine ha origine nel contesto dell’arte, dove serve per descrivere la storia di un’opera d’arte e per individuare chi l’ha creata e a chi va attribuita.

Nell’informatica indica i metadati che descrivono le origini di un dato (es. dati di input, risultati intermedi).

Tracciare le informazioni di provenienza può essere utile per analisi e debug successivi per risalire ai file che hanno generato un certo risultato (backward) o i file generati da un input/implementazione errata (forward).

I metadati possono essere modellati come un grafo diretto aciclico (DAG), dove *A* → *B*

indica che A è stato usato per derivare B



La Figura 2-1 illustra la provenienza per la seguente query: SELECT *sum*(*a*) FROM *T* WHERE *a >* 10 rappresentata attraverso un albero.

La provenienza consiste di operatori, input, risultati intermedi e finali, dipendenze. Una volta che il grafo è stato creato, è possibile interrogarlo.

LINEAGE

La data lineage è un sottoinsieme della data provenance, che si concentra sulle dipendenze tra i record di input e di output di una data elaborazione. Spesso i due termini sono utilizzati in modo intercambiabile.

# Titian

Autori: Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, Tyson Condie

### Informazioni

Titian è una libreria nata per tracciare la data provenance su Apache Spark v1.2.1 (ora esteso a Spark 2.1.1)

L'obiettivo è identificare velocemente i dati di input responsabili di un bug o di un risultato inaspettato (outlier).

Il modello di programmazione dei sistemi DISC (Data-intensive scalable computing) viene compilato in un grafo diretto aciclico (DAG) di operatori. L'operatore radice consuma i dati da una sorgente di input, gli operatori successivi utilizzano i dati intermedi.

La scalabilità è ottenuta partizionando i dati e assegnando task a vari nodi.

Gli approcci precedentemente adottati per tracciare la data provenance (Newt e RAMP) presentano dei limiti:

* usano sistemi di memorizzazione esterni (DBMS, HDFS) per le informazioni sulla provenienza
* le query di provenienza vengono effettuate su un'interfaccia di programmazione esterna
* non forniscono supporto per vedere dati intermedi o replicare i passi di elaborazione su di essi

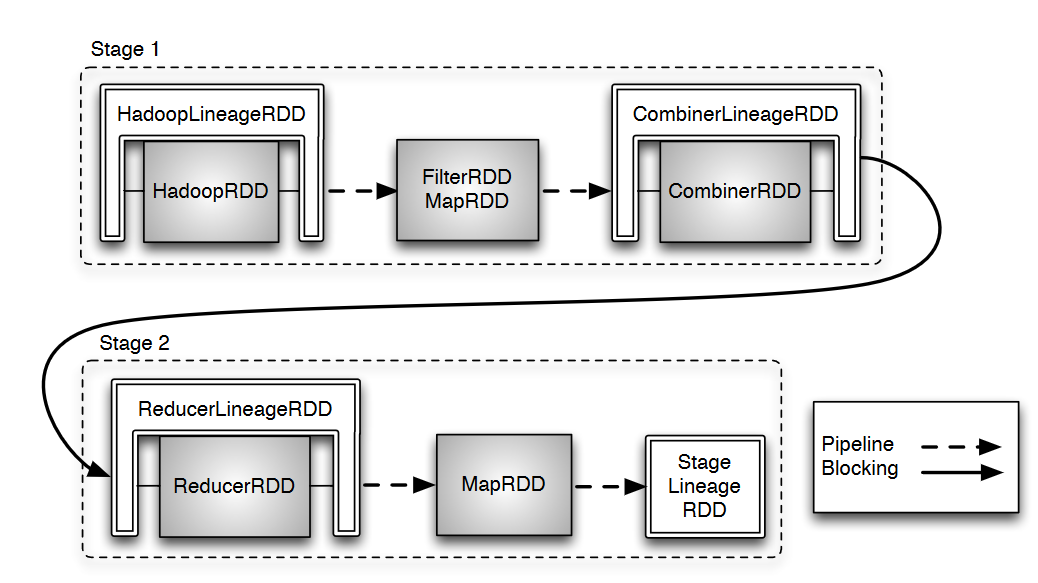
Spark si basa sugli RDD (resilient distributed dataset) su cui vengono effettuate le trasformazioni e le azioni per elaborare i dati.  
I programmi driver di Spark operano sugli RDD mediante riferimento. I riferimenti agli RDD valutano in modo pigro le trasformazioni restituendo un riferimento ad un nuovo RDD, e soltanto le azioni innescano la valutazione del riferimento e di tutte le trasformazioni che lo precedono e operano su di esso.

Spark traduce una serie di trasformazioni sugli RDD in un DAG di fasi (stage):

* una fase contiene una sotto-sequenza di trasformazioni che precedono uno *shuffle step*1 (che ripartiziona i dati)

Lo scheduler esegue gli stage in ordine topologico: ogni stage è eseguito da un task che applica le sue trasformazioni su ogni partizione dell'input.

Infine i valori risultanti dall'ultima azione vengono collezionati da ogni task e restituiti al driver, che allora può iniziare una nuova serie di trasformazione terminanti in un'azione.



Sistema Operativi supportati: Unix-like (i test sono stati effettuati su Ubuntu 20.04, Focal Fossa)

### Configurazione

* clonare il progetto disponibile all’url https://github.com/maligulzar/bigdebug (branch: titian-2.1)
* unzip della cartella bigdebug-titian-2.1.zip
* spostarti sulla cartella bigdebug-titian-2.1
* aprire un terminale e fare la build come root tramite build/mvn -DskipTests clean package
* assicurarsi che la build sia conclusa con successo (build success), in genere ci vogliono 6-7 minuti

1operazioni che possono causare uno shuffle: cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, sortByKey, distinct, intersection, repartition, coalesce

### Esercizi

Esercizi interattivi

È possibile trovare del codice di riferimento presso bigdebug-titian-2.1/lineage/src/test/scala/org/apache/spark/lineage/LineageSuite.scala

Es. 1

* creare un file grep.scala alla radice del progetto, avente il seguente contenuto

import org.apache.spark.SparkContext

import org.apache.spark.lineage.LineageContext

import org.apache.spark.util.collection.CompactBuffer

val logFile = "lineage/src/test/resources/README.md"

val lc = new LineageContext(sc)

lc.setCaptureLineage(true)

val lines = lc.textFile(logFile, 2)

val result = lines.filter(line => line.contains("spark"))

print(">>>result's size: "+result.collect().size+"\n")

lc.setCaptureLineage(false)

var linRdd = result.getLineage()

linRdd = linRdd.filter(\_ == 0L)

print(">>>linRdd: "+linRdd.collect()(0)+"\n")

linRdd = linRdd.goBack()

print(">>>linRdd: "+linRdd.collect()(0)+"\n")

print(">>>linRdd: "+linRdd.show.collect()(0)+"\n")

linRdd = linRdd.goNext()

print(">>>linRdd: "+linRdd.collect()(0)+"\n")

* eseguire shell presso la root del progetto: bin/spark-shell --driver-memory 4g
* :load grep.scala

Output atteso:

Loading grep.scala...

import org.apache.spark.SparkContext

import org.apache.spark.lineage.LineageContext

import org.apache.spark.util.collection.CompactBuffer

logFile: String = lineage/src/test/resources/README.md

lc: org.apache.spark.lineage.LineageContext = org.apache.spark.lineage.LineageContext@120ed76a

lines: org.apache.spark.lineage.rdd.Lineage[String] = lineage/src/test/resources/README.md MapPartitionsLRDD[76] at MapPartitionsRDD at MapPartitionsLRDD.scala:28

result: org.apache.spark.lineage.rdd.Lineage[String] = MapPartitionsLRDD[77] at MapPartitionsRDD at MapPartitionsLRDD.scala:28

>>>result's size: 11

linRdd: org.apache.spark.lineage.rdd.LineageRDD = LineageRDD[80] at RDD at LineageRDD.scala:39

linRdd: org.apache.spark.lineage.rdd.LineageRDD = LineageRDD[82] at RDD at LineageRDD.scala:39

>>>linRdd: (0,(0,9))

linRdd: org.apache.spark.lineage.rdd.LineageRDD = LineageRDD[84] at RDD at LineageRDD.scala:39

>>>linRdd: ((0,9),0)

<http://spark.apache.org/>

>>>linRdd: <http://spark.apache.org/>

linRdd: org.apache.spark.lineage.rdd.LineageRDD = LineageRDD[96] at RDD at LineageRDD.scala:39

20/09/18 12:30:36 WARN Executor: 1 block locks were not released by TID = 31:

[rdd\_78\_1]

>>>linRdd: (0,9)

Nell’esercizio proposto abbiamo filtrato le righe contenenti la parola spark e siamo risaliti fino alla prima riga nell’input contenente tale parola.

Essa viene mostrata tramite invocazione di show e collect su un oggetto di tipo lineageRDD

Una collect senza show sugli oggetti lineageRDD invece restituisce gli offset dei record di cui abbiamo costruito la lineage che soddisfano la condizione imposta.

Esercizi Standalone

* clonare il progetto https://github.com/maligulzar/bigdebug (branch titian-2.1) e farne l’unzip
* da shell build/mvn -DskipTests clean package
* Fare il download dell’IDE Intellij e avviare IntelliJ
* file>new>project>scala>IDEA

scala-sdk-2.11.8

java jdk 1.8 (io ho usato 1.8.0\_265)

* dargli un nome, per es. titian
* dal progetto bigdebug-titian-2.1 di cui abbiamo fatto il download copiare la cartella assembly>target>scala-2.11>jars alla root del progetto titian
* file>project structure>modules>add>jars or directories
* aggiungere il puntamento a questa cartella
* creare sotto src un package main
* src/main: click destro>new>Scala class>Object

inserire il codice a seguire

import org.apache.spark.SparkContext

import org.apache.spark.lineage.LineageContext

import org.apache.spark.lineage.LineageContext.\_

import org.apache.spark.util.collection.CompactBuffer

val logFile = "wcount.txt"

var lc = new LineageContext(sc)

lc.setCaptureLineage(true)

val file = lc.textFile(logFile, 2)

val pairs = file.flatMap(line => line.trim().split(" ")).map(word => (word.trim(), 1))

val result = pairs.reduceByKey(\_ + \_)//.asInstanceOf[Lineage[(String,Int)]]

val r4=result.collect

lc.setCaptureLineage(false)

val l5=result.getLineage()

val l6=l5.goBack()

val r4=l6.collect

val rr3=l6.show

print("\n")

val l7=l6.goBack()

val r5=l7.collect

val rr4=l7.show

creo un lineagecontext passandogli come parametro uno sparkcontext

imposto la variabile globale captureLineage a true per indicare che sto iniziando a tracciare la lineage.

TapLRDD è una classe che estende RDD e segna di volta in volta l'RDD corrente e una relazione uno-a-uno tra le partizioni dell'RDD genitore e dell'RDD corrente (figlio)

goback permette di controllare l’rdd iniziale della stage precedente

* le posizioni dei lineage precedenti vengono mantenute in uno stack prevLineagePosition
* dependencies è una proprietà degli RDD che permette di risalire agli RDD da cui esso dipende
* salvo l'ultima operazione effettuata (backward) in lastOperation
* navigo verso la stage precedente se esiste e aggiorno le dependencies

Quando intendo terminare la cattura della lineage imposto captureLineage a false

Poiché

* Titian registra le informazioni relative agli RDD e alle operazioni eseguite su di essi tramite un LineageContext, una classe wrapper di SparkContext
* Le operazioni di data pre-processing operano su una SparkSession
* MLlib non è stato adattato per tracciare le informazioni di provenienza (non fa parte di Titian o di BigDebug, progetto affine)

Dopo aver utilizzato la librerie per vederne la fattibilità ho escluso la possibilità di applicare data provenance a problemi di machine learning tramite Titian.

Purtroppo Titian cattura la provenienza a livello di stage e non è possibile quindi utilizzarlo per avere informazioni di provenienza sul data pre-processing. L’utilità è rappresentata dal poter tracciare la provenienza all’indietro quando vengono coinvolte più stage.

# Scorpion

Autore: Eugene Wu

### Informazioni + Esercizi

Outlier: valori aggregati o sottogruppi di punti che si comportano diversamente dalle aspettative dell’utente.

Il valore di un outlier può dipendere da migliaia o milioni di record di input. Restituire in modo naive tutti i record di input non è informativo: occorrono tecniche per sintetizzare e ridurre le informazioni di provenienza necessarie.

Scorpion è un tool per generare ipotesi e spiegare gli outlier nel risultato di query SQL che usano aggregazioni. Identifica sottoinsiemi dell’input che sono più correlati ai valori degli outlier indicati dall’utente.

L’“analisi del perché” (why-analysis) cerca di scoprire gli outlier descrivendo le proprietà comune dei dati di input che hanno generato quegli outlier.

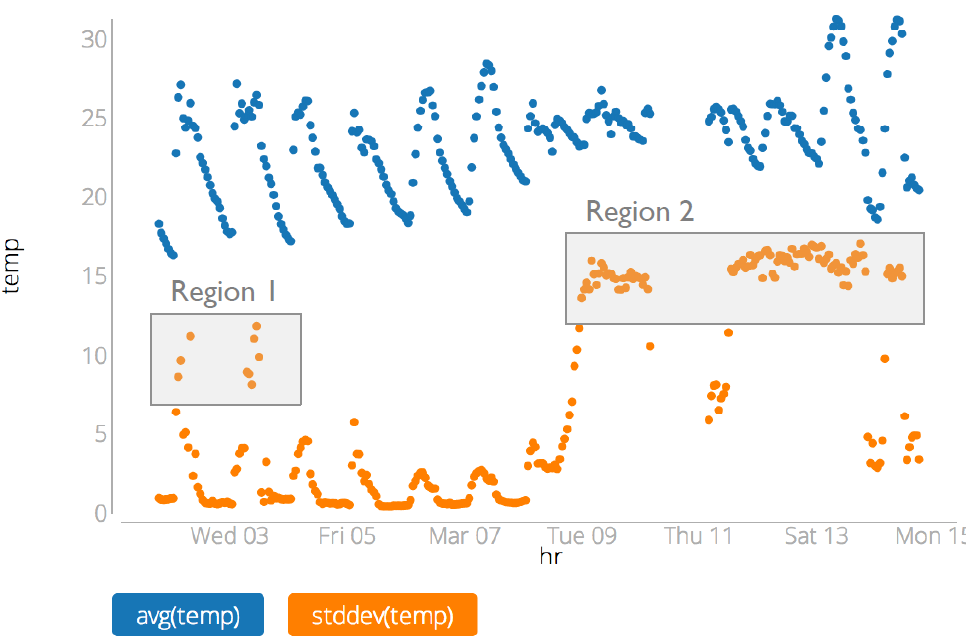
Il seguente esempio rappresenta una visualizzazione dei dati relativi ai sensori di Intel per le temperature. Ogni punto è un aggregato (media o deviazione standard) dei dati di 60 sensori. Mentre la deviazione standard fluttua pesantemente, la temperatura smette di oscillare. Per spiegare questo comportamento occorre trovare un predicato booleano che applicato ai dati di input prima del calcolo dell’aggregazione farà sembrare gli outlier normali avendo minimo impatto sui punti che l’utente ha indicato come normali (hold-out).

La regione blu nell’immagine seguente è rappresentata dalla query

SELECT avg(temp),time

FROM sensors

GROUP BY time



* La regione 1 è dovuta a dati di sensori vicini alle finestre che si scaldano a mezzogiorno
* La regione 2 è relativa ad un sensore scarico (basso voltaggio) che inizia a produrre letture errate

Poiché questi fatti non sono ovvi spesso si ricorre ad un’ispezione manuale dei dati.

Per risolvere il problema occorre risolvere 3 sotto-problemi:

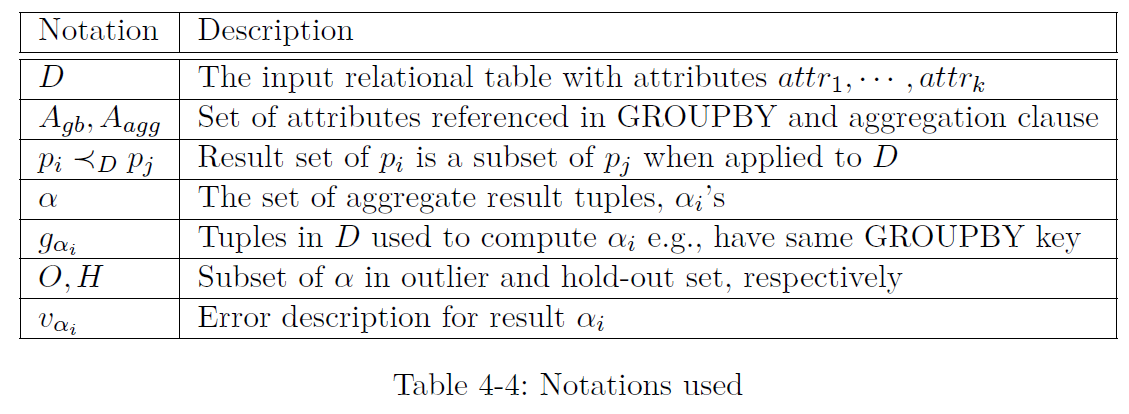
* Backward provenance: risalire ai punti di input utilizzati per calcolare gli outlier. Assumiamo che input e output siano relazioni e che gli output siano generati da group-by e aggregazioni sugli input.
* Sottoinsieme responsabile: trovare le tuple di input che hanno contribuito di più agli outlier
* Generazione di un predicato sugli attributi di input per determinare il sottoinsieme responsabile

Scorpion usa la sensitivity analysis.

Per determinare il predicato occorre confrontare come i candidati si influenzano: qui interviene una funzione di punteggio (scoring) che induce un ordinamento parziale sullo spazio dei predicati.

Comprendere le anomalie genera diverse domande che orientano le analisi successive.

In tabella vengono definite le notazioni utilizzate

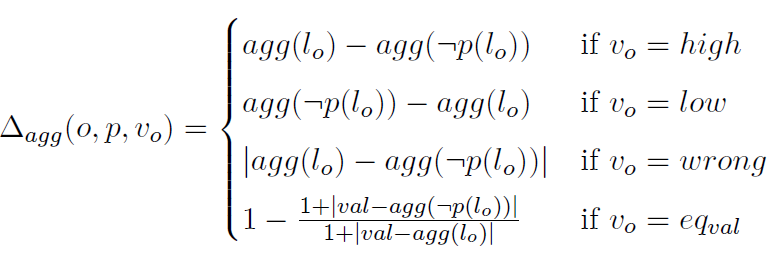


Definisco ora la funzione che determina l’influenza di un predicato sugli outlier infagg(•)

Essa si basa sull’analisi di sensitività: data una funzione *y* = *f*(*x*1*,* ・ ・ ・ *, xn*) l’influenza di *xi* è definita dalla quantità di cambiamenti esercitati sull’output da un cambiamento in *xi*, ovvero la derivata parziale Δ*y/*Δ*xi*

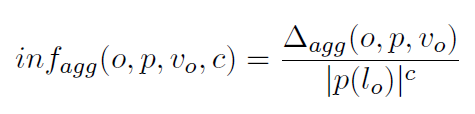
Dato un insieme di tuple lo e un output o, l’influenza di un predicato p su o dipende dalla differenza tra il risultato originale e il risultato aggiornato dopo aver eliminato p(lo) da

lo.



Prende in considerazione le descrizioni degli errori, ovvero se gli outlier sono troppo alti/bassi. Infatti l’utente quando specifica gli outlier e gli holdout caratterizza la differenza che esiste tra questi due insiemi.

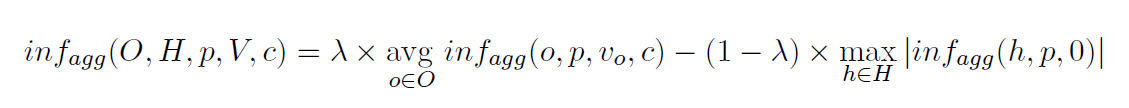
Per un singolo outlier la funzione di influenza è



p(lo) è un termine di regolarizzazione per evitare soluzioni banali in cui p=True che massimizzano lo score di funzioni di aggregazione come la COUNT.

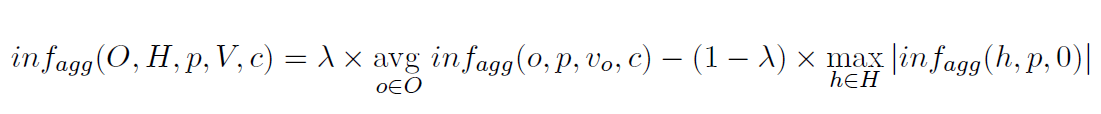
L’esponente c>=0 a denominatore controlla il tradeoff rappresentato dal mantenere la dimensione di p(lo) piccola e massimizzare i cambiamenti desiderati nell’output. C controlla quanto aggressivamente Scorpion dovrebbe ridurre il risultato: aumentare C produce predicati molto più selettivi.

Poiché l’utente spesso seleziona più outlier O e più hold-out (valori considerati normali), la funzione viene estesa per gestire più risultati.



λ viene utilizzata per bilanciare i due goal in competizione:

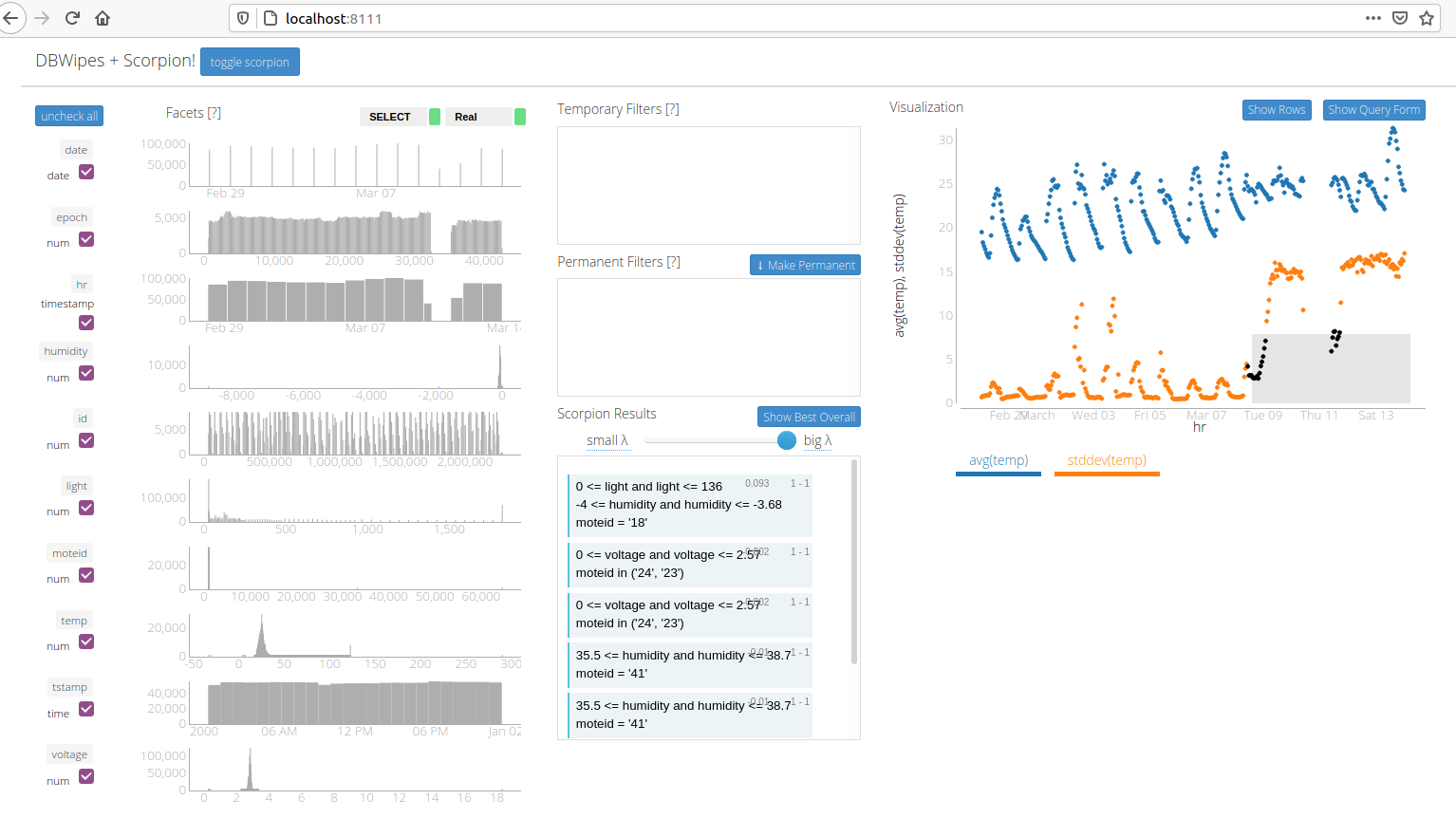
* Influenzare gli outlier
* Non influenzare gli holdout



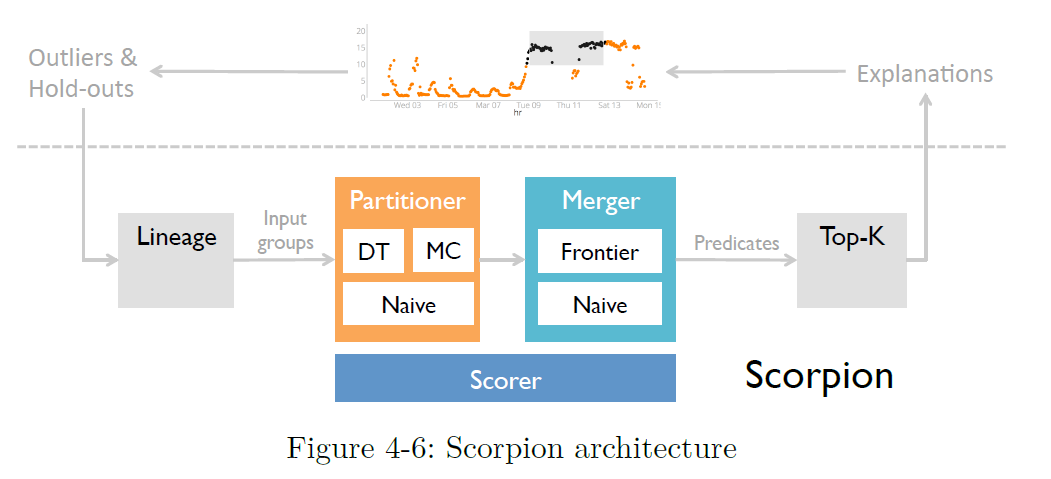
Sono state fatte delle assunzioni:

* Group-by assumption: senza un aggregatore ogni risultato dipende banalmente su un singolo record di input
* Subquery assumption: le subquery non sono concesse poiché in esse la funzione che misura la distanza non può essere valutata in modo non ambiguo
* Join assumption: i join non sono concessi per ragioni di convenienza (rendere il testo semplice) e per ragioni di efficienza (un record di input può contribuire più volte ad un risultato e può influenzare più risultati)

A seguire una spiegazione dei dati da me segnalati come outlier porta Scorpion a determinare come cause più probabili quelle dovute a certi livelli di luce ed umidità o di voltaggio. Questo rappresenta correttamente le motivazioni per cui i dati sono degli outlier.



**Architettura**



Gli utenti possono selezionare un database ed eseguire query con aggregazioni i cui risultati sono visualizzati con dei grafici. A quel punto possono arbitrariamente selezionare dei risultati, etichettarli come outlier o holdout, specificare attributi che devono essere ignorati nella ricerca dei predicati e mandare la richiesta al backend di Scorpion. Infine possono cliccare le spiegazioni dei risultati e ridisegnare l’output aggiornato che non presenta più gli outlier di input.

Scorpion usa il componente della Lineage per calcolare la provenienza dei risultati etichettati.

La lineage e gli input sono passati al Partitioner che sceglie l’algoritmo di partizionamento sulla base delle proprietà di un aggregato e genera un ranked list di predicati, ciascuno con uno score che rappresenta la sua influenza stimata. Poiché l’algoritmo di partizionamento spesso sovra-partiziona il dataset, Scorpion esegue una fase di merge (Merger) che fa il merge di predicati simili con l’obiettivo di incrementarne l’influenza.

Partitioner e Merger mandano i predicati candidati allo Scorer che calcola l’influenza, il cui costo domina il costo complessivo del sistema poiché deve eliminare dalla lineage di ogni risultato le tuple che matchano con il predicato e rieseguire l’aggregazione sulla lineage aggiornata.

Infine i predicati più influenti sono restituiti e visualizzati.

Andando a vedere il codice sorgente di Scorpion (https://github.com/sirrice/scorpion), presso scorpion/sigmoid/merger.py possiamo vedere la funzione che calcola l’influenza di un predicato in cui:

* good\_inf è l’influenza esercitata sui data point indicati come hold-out
* bad\_inf è l’influenza esercitata sui data point indicati come outlier

def influence\_from\_mtuples(self, merged, intersecting\_clusters):

bad\_states, good\_states = self.get\_states(merged, intersecting\_clusters)

if not sum(map(bool, bad\_states)):

return None

@instrument

def get\_influences(self, efs, states, master\_states,c ):

infs = []

for ef, state, mstate in zip(efs, states, master\_states):

if state:

influence = ef.recover(ef.remove(mstate, state))

if state[-1]\*\*c:

influence = influence / (state[-1]\*\*c)

infs.append(inluencef)

return infs

bad\_efs = self.learner.bad\_err\_funcs

good\_efs = self.learner.good\_err\_funcs

bad\_infs = get\_influences(self, bad\_efs, bad\_states, self.learner.bad\_states, self.learner.c)

good\_infs = map(abs, get\_influences(self, good\_efs, good\_states, self.learner.good\_states, 0) or [])

if not bad\_infs:

return -1e10000000

bad\_inf = bad\_infs and np.mean(bad\_infs) or -1e100000000

good\_inf = good\_infs and np.mean(good\_infs) or 0

l = self.learner.l

return l \* bad\_inf - (1. - l) \* good\_inf

### Configurazione

Requisiti: SO unix-like (ho provato anche su Windows 10 ma ci sono dei problemi a tempo di esecuzione di Scorpion con la libreria matplotlib).

Installazione su Ubuntu 20.04 focal2

Prima di tutto, dopo aver installato python e pip (assicurarsi che la versione in uso di python sia la 2.7 con python -V) provare quanto indicato dall’autore su github https://github.com/sirrice/scorpion

* pip --no-cache-dir install scorpion

Se si incontra il seguente errore: command errored out with exit status 1: python setup.py egg\_info

Seguire questi passi

1. Installare python 2.7 e pip, assicurarsi che la versione in uso di python sia la 2.7 (python -V)
2. pip install --upgrade pip setuptools wheel
3. installare anaconda
4. conda create -n virt python=2.7.0
5. sudo apt install virtualenv build-essential python2.7-dev libspatialindex-dev build-essential g++
6. conda install -c free orange
7. cercare su pip dbwipes, andare su download files, download di dbwipes.tar.gz
8. unzip del tar scaricato e modificare i package\_data di setup.py:

package\_data = {

'static': ['dbwipes/static'],

'templates': ['dbwipes/templates']

}

1. posizionarsi sulla cartella contenente setup.py di dbwipes
2. python setup.py install
3. sudo apt update
4. sudo apt install libdb-dev
5. sudo apt install postgresql
6. psql --version #psql (PostgreSQL) 12.4 (Ubuntu 12.4-0ubuntu0.20.04.1)
7. sudo systemctl is-active postgresql
8. sudo systemctl is-enabled postgresql
9. sudo systemctl status postgresql
10. impostare a trust la connessione su localhost in hba\_conf di postgresql
11. pip install scorpion flask-compress scorpionsql
12. wget "https://www.dropbox.com/s/glutiyu2uju4ijq/intel.ddl?dl=0"
13. Modificare il file intel.ddl\?dl=0 appena il download è concluso sostituendo tutte le occorrenze di ewu con il nomeruolo scelto (es. giacomo)

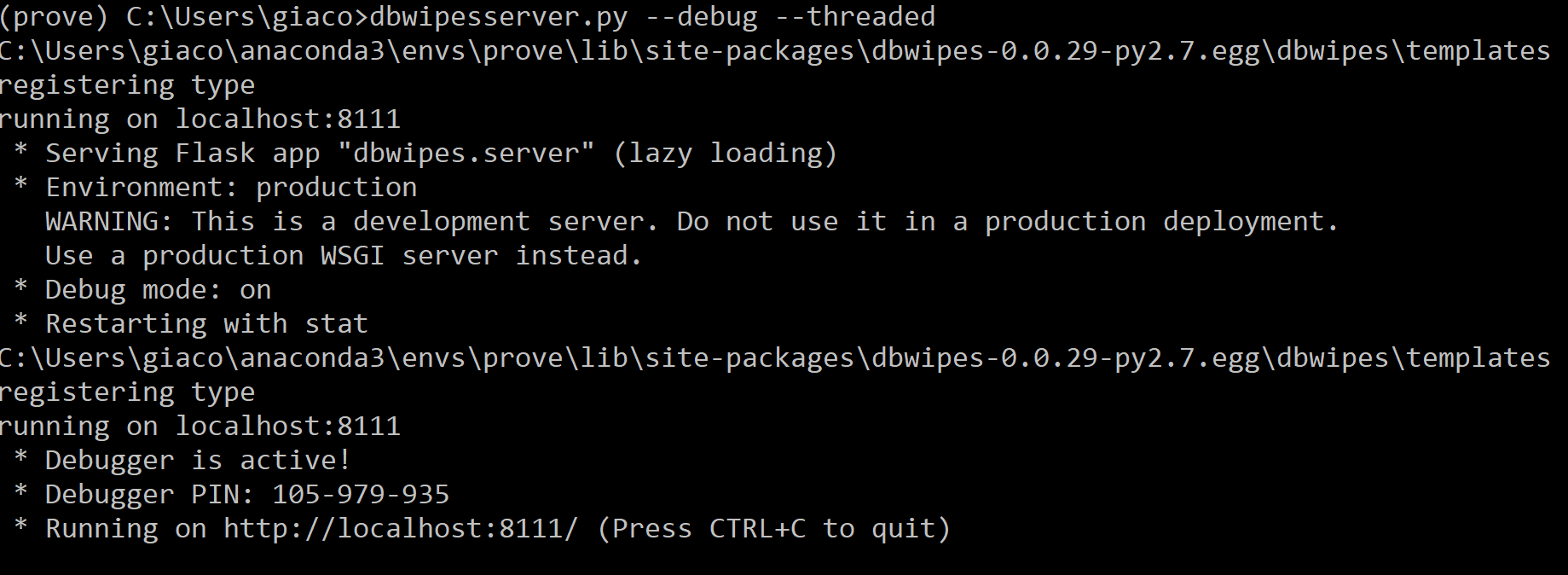
Una volta che postgres è installato:

* sudo su - postgres
* psql
* CREATE ROLE nomeruolo WITH SUPERUSER CREATEDB CREATEROLE LOGIN ENCRYPTED PASSWORD '1234';

Al posto di nomeruolo indicare un nome per il ruolo che si sta creando (lo stesso vale per tutti gli altri riferimenti a nomeruolo successivi; in questo caso per es. CREATE ROLE giacomo WITH SUPERUSER CREATEDB CREATEROLE LOGIN ENCRYPTED PASSWORD '1234';)

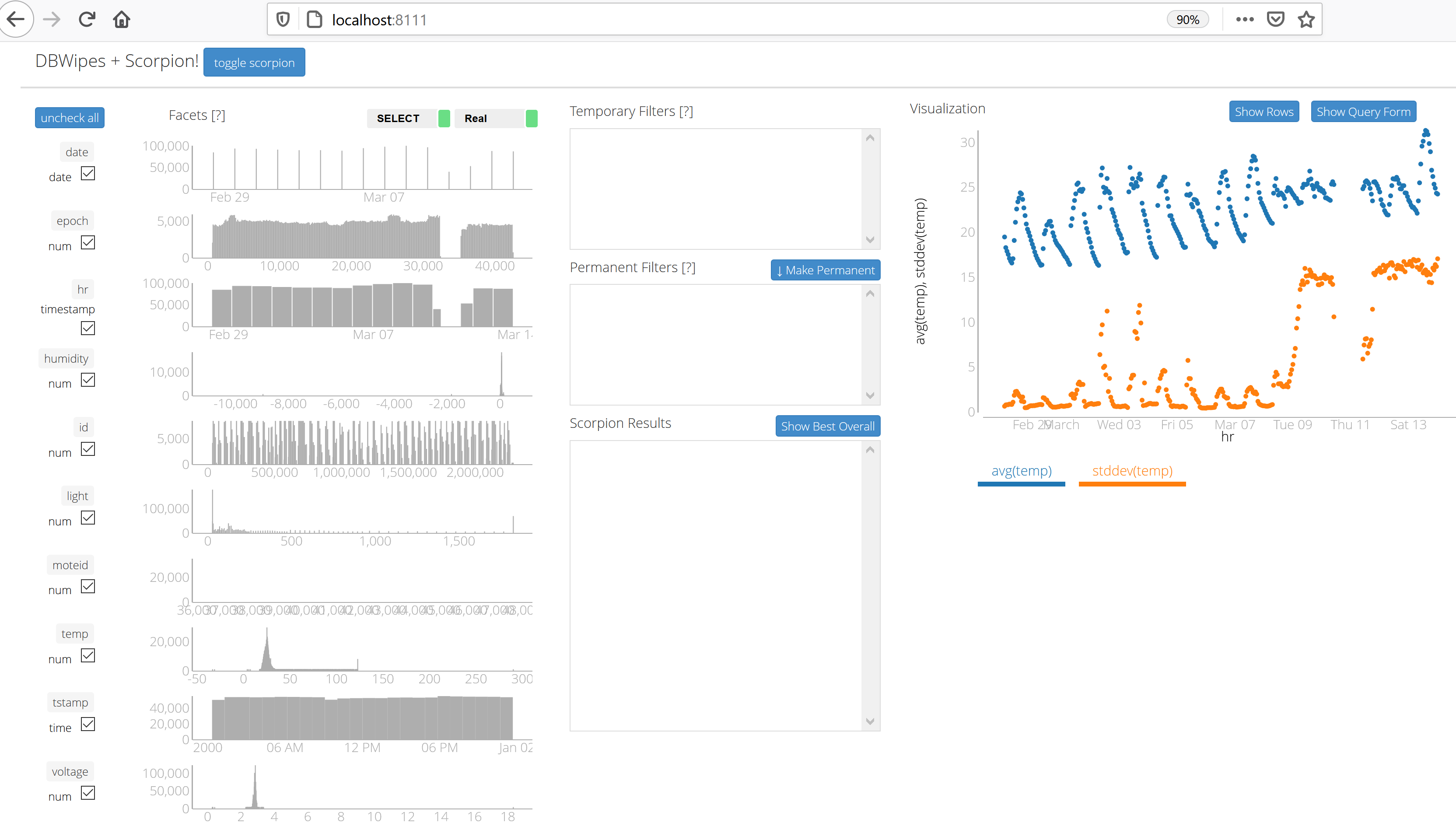
* CREATE DATABASE intel OWNER nomeruolo;
* CREATE DATABASE cache OWNER nomeruolo;
* CREATE DATABASE status OWNER nomeruolo;
* Uscire da psql tramite ctrl+c ed eseguire psql -f intel.ddl\?dl=0 intel
* Digitare sulla shell: dbwipesserver.py --debug --threaded

Si vedrà qualcosa di simile a questo

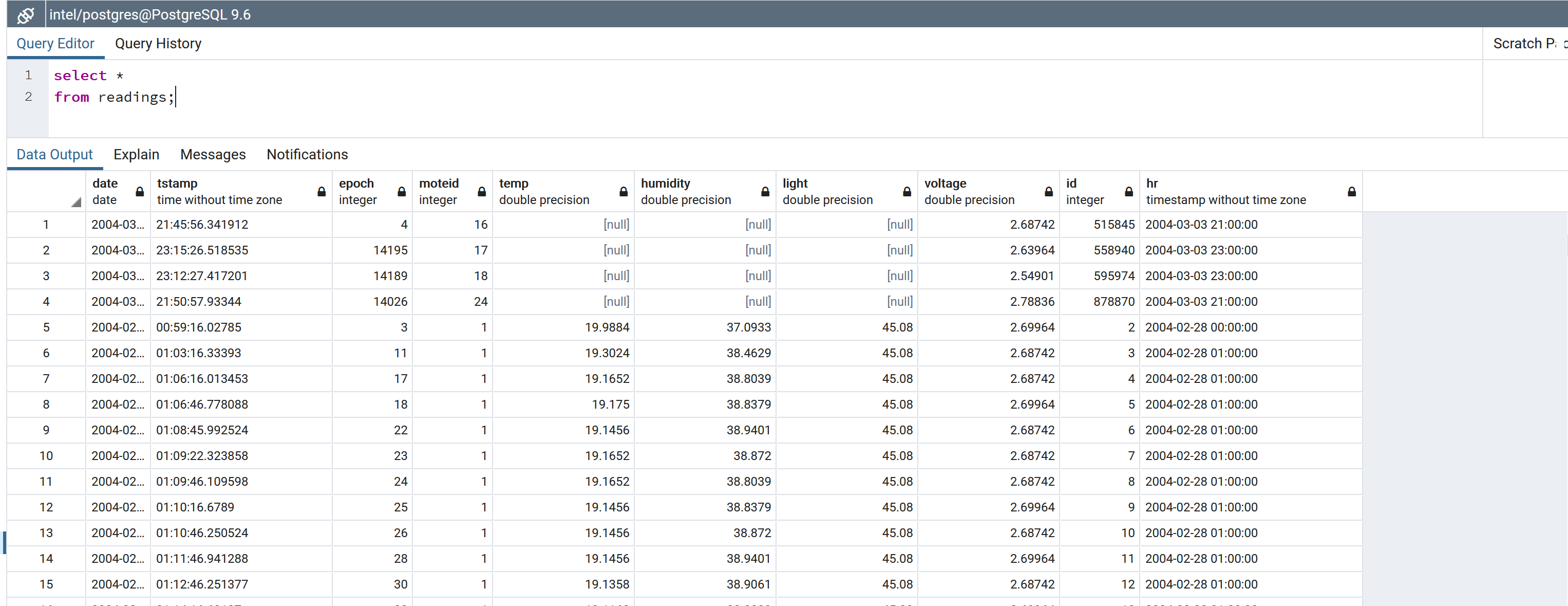


* Aprire ora una finestra su un browser: http://localhost:8111/

Si vedrà



Se si usa pgAdmin è possibile vedere i dettagli della tabella readings del database intel



Comandi utili postgres:

* Interrompere postgres: sudo systemctl stop postgresql.service
* Avviare postgres: sudo systemctl start postgresql.service

Comandi utili psql:

* \l per mostrare una lista dei database
* elenca le tabelle:\dt
* \du per elencare gli utenti
* q per uscire

Alcuni errori che si possono incontrare

* Can't find a local Berkeley DB installation.

Si risolve con sudo apt install libdb-dev

* ERROR: dbwipes 0.0.29 requires flask-compress, which is not installed.

ERROR: dbwipes 0.0.29 requires scorpionsql, which is not installed.

Si resolve con pip install flask-compress scorpionsql

* Dbwipes non termina, rimane in ascolto

sudo apt install net-tools

sudo netstat -nlp | grep 8111 #per recuperare

sudo kill -9 pid #al posto di pid indicare l’id del processo ancora in ascolto sulla porta 8111

* Se il grafico non viene visualizzato è possibile che non si abbiamo i privilegi sul database intel

GRANT ALL PRIVILEGES ON DATABASE intel to nomeruolo;

Alcuni prerequisiti

* libspatialindex-dev#prerequisito per tree
* python2.7-dev#sol. per avere psycop2
* g++#prereq. orange (|sudo apt install build-essential)

# Pre-processing Provenance

Autore: Giulia Simonelli

### Informazioni

Le pipeline per il pre-processing possono essere viste come una composizione di operatori, che quindi vengono formalizzati e categorizzati in:

* Data reduction
* Augmentation
* Transformation

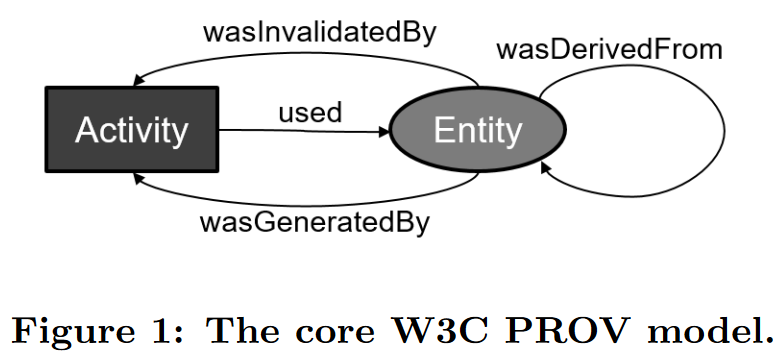
In una classe Provenance realizzata in Python vengono definite le funzioni per catturare la provenienza per gli specifici operatori.

I file di provenienza possono poi essere salvati in un database MongoDB.

* Un’entità rappresenta un element d di un dataset D univocamente identificata da D e dalle sue coordinate (indice di riga e feature) in D
* Un’attività è un’operazione di manipolazione dei dati di un dataset

Inoltre per ogni elemento d di un dataset D’ generato da un’operazione o su D:

* d è generato da o
* d deriva da un insieme di elementi in o
* o può aver utilizzato degli elementi d
* d potrebbe essere stato invalidato da o (ad es. una feature che è stata rimossa da un’operazione di riduzione di dimensionalità)



Data Reductions

* Feature Selection
* Instance Selection

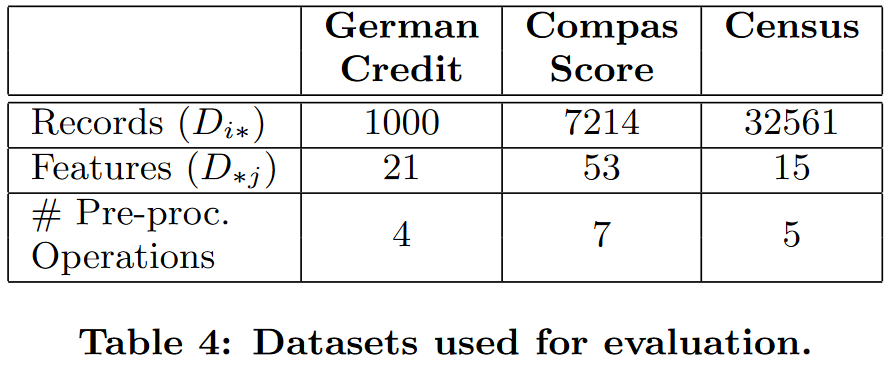
Data Transformations

* Data repair
* Binarization
* Normalization
* Discretization
* Imputation

Data augmentations

* Space Transformation
* Instance Generation
* String Indexer
* One-Hot Encoder

La valutazione viene fatta su 3 dataset distinti, a dimensione via via crescente. Il dataset più grande è pari a 3.8 MB



Inoltre la scelta della grana per la provenienza che viene catturata impatta prestazioni e capacità del sistema di rispondere alle query. Considerando gli approcci in grado di catturare provenienza a grana fine, che sono in grado di rispondere a tutte le query che il paper ha definito:

* FP genera un unico documento monolitico durante l’esecuzione dell’intera pipeline di operatori
* FP+ genera diversi frammenti di provenienza che possono essere poi messe insieme e collegati a tempo di interrogazione; grazie ad esso ne beneficiano le prestazioni al momento della cattura della provenienza

Purtroppo allo stato attuale non è supportata la registrazione della data provenance in maniera distribuita, utilizzando cluster, ma è svolto in locale su macchina singola, mediante le librerie pandas e numpy. Una buona alternativa, se non spark, qui sarebbe stata utilizzare dask per sfruttare le elaborazioni di pandas in maniera distribuita.

Allo stato attuale il codice non è in grado di catturare informazioni di provenienza sulle pre-elaborazioni di file più grandi della RAM del computer, dispositivo o server su cui esso viene eseguito.

### Configurazione

SO supportati: unix-like e Windows

Ho fatto i test su Windows 10

Prerequisiti: Python, pip

* pip install jupyter ipynb pandas numpy prov pydot pymongo

Inoltre è necessario installare graphviz (1) e mongoDB (2)

(1): scaricare il plugin, aggiungere il path alla cartella bin di graphviz nelle variabili di ambiente, usare la powershell e accettare l’accordo di licenza con pydot -c

Per eseguire il codice:

* clonare il progetto disponibile all’indirizzo https://github.com/GiuliaSim/Provenance
* fare l’unzip del progetto
* aprire un prompt dei comandi digitando cmd nella barra di ricerca di Windows
* posizionarsi nella cartella ProvenanceMaster
* eseguire il comando jupyter notebook <nomefile ipynb>

### Esercizi

Caso di studio: Census Cleanup

Eseguire gli step 1-4

jupyter notebook CensusCleanup\_prov.ipynb

Obiettivi del pre-processing:

* Eliminare gli spazi da 9 colonne
* Sostituire il carattere ? con NaN
* One-hot encoding di 7 colonne

Rendere binari (0,1) sesso ed etichetta

Cancellare la colonna fnlwgt

Definisco una funzione timing e uso un decoratore @timing sulle funzioni per registrare il tempo di esecuzione all’interno di un log.



Includo le librerie necessarie ed effettuo il download del dataset dall’indirizzo https://raw.githubusercontent.com/vladoxNCL/fairCorrect/master/Datasets/adult.data

Il dataset è costituito da 32561 record organizzati in 15 colonne:

'age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'capital-gain', 'capital-loss', 'hours-per-week', 'native-country', 'label'

Creo un nuovo documento di provenienza

Inizializzo le attività, relazioni, entità con delle liste vuote, imposto il nr di operazione a -1 e l’istanza pari a output-1.

Nota: registrando il numero di operazione tengo traccia dell’ordine in cui sono eseguite le operazioni, quindi l’utente può arbitrariamente scegliere in che ordine eseguire le varie parti della pipeline di preprocessing.

Imposto alcuni attributi pari a quelli del dataframe chiamante: colonne, indici (un intervallo di interi da 0 al numero di record/righe) correnti, m (numero record) e n (numero colonne/attributi) correnti. Quindi avrò una shape pari a mxn.

Creo le entità passandogli il nome della cartella di input, mediante il metodo create\_entity richiamato all’interno di un doppio ciclo for (sulle righe e sulle colonne).

Ogni entità ha:

* un identificatore entity:<*alfrand>* dove <*alfrand*> è un alfanumerico creato in modo casuale
* un valore pari a quello dell’elemento acceduto tramite indici di riga e colonna
* un attributo

Ogni attributo di entità ha un identificatore, un nome, un valore, un indice e un numero di istanza (-1)

Creata l’entità, viene aggiunta alla lista delle nuove entità

Preparo la cartella dei risultati result/<nomeProgetto> o result/<*timestamp>* se il nomeProgetto non è specificato

Salvo le entità create in dei file json sotto result/<nomeProgetto>/input/entities<\_$nrChunk> tramite la funzione save\_json\_prov

In pratica le mxn entità vengono suddivise in vari file.

In questa fase non ci sono attività né relazioni.

**Operazione 0: Eliminare gli spazi da 9 colonne**

L’operazione è una trasformazione di attributi.

Le colonne in esame sono: 'workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country', 'label'.

Itero sulle colonne e ne privo i valori degli spazi di inizio e fine

Inizializzo entità, relazioni e attività correnti con delle liste vuote, incremento il numero di operazione

Considero le entità create, le colonne totali, gli indici e le colonne sotto esame

Per ogni colonna sotto esame creo un attività

Ogni attività è formata da id e attributi.

Ogni attributo di attività possiede:

* un nome di funzione (in questo caso trasformazioni di attributi)
* un nome di feature
* una descrizione: es. rimuovere gli spazi dai nomi
* un numero di operazione (0)
* other\_attributes se presenti

Aggiungo l’attività alla lista delle attività correnti

Creo una nuova entità legata all’attributo in corso di modifica, registrando l’entità origine, il valore e altre informazioni.

Lego il tutto mediante 4 relazioni:

* GENERAZIONE: l’entità risultato genera l’attività
* USO: l’attività usa l’entità
* DERIVAZIONE: la nuova entità (generata) deriva da un’entità origine (d’uso)
* INVALIDAZIONE: l’entità è invalidata (es. rimozione di un attributo) dall’attività

Aggiorno le relazioni correnti

Salvo su file json: results/nomeProgetto/input/relations<\_nrRelazione>

**One-hot encoding**

Itero sulle colonne workclass e race.

Per ogni colonna, uso numpy.unique\_val per ricavare i valori unici in essa contenuti.

Trasformo le variabili categoriche in vettori di 0-1 (one-hot encoding) di dimensione m mediante il metodo pandas.get\_dummies

Concateno poi le nuove colonne ottenute con il dataframe originale: arriverò a un totale di 22 colonne.

Registro la provenienza tramite get\_prov\_onehot\_encode

Creo delle entità per ogni singolo elemento mxn.

Per ogni riga recupero l’identificatore dell’entità relativa alla colonna/attributo modificato.

Per ogni colonna creo un’attività OneHot Encoding

Creo la relazione d’uso tra l’attività e l’entità

Se l’attributo è stato sottoposto a onehot encoding (non si trova più nella lista delle nuove colonne), allora creo la relazione “l’entità è stata invalidata dall’attività”

Tramite un triplo ciclo for recupero il nome della feature cambiata per ogni colonna 0-1 in esame e creo una nuova entità corrente e le due relazioni:

* l’entità corrente ha generato l’attività corrente
* l’entità corrente deriva da un’entità genitrice

Aggiungo la nuova entità alla lista delle entità.

Le colonne non cambiate infine vengono iterate.

Considero ogni colonna e la sottopongo a riarrangiamento: vengono scalate tutte verso sinistra, cambiando la vecchia posizione/indice con la nuova posizione/indice.

Salvo in file json.

**Binarizzazione di sex e label**

Anche qui è un’operazione di feature transformation (vd operazione 0)

**Riduzione di dimensionalità**

Getto via tramite drop la colonna fnlwgt perché poco significativa per le analisi.

Anche qui viene fatta l’inizializzazione e viene creata una nuova attività Dimensionality reduction con i nomi delle colonne rimosse dal dataframe.

Creo una relazione di invalidazione tra l’entità e la nuova attività

Aggiorno le entità di input e di output

<entities\_in = np.delete(entities\_in, list(delIndex), axis=0)?;

delIndex è una lista vuota>

aggiorno i valori correnti e salvo su file json.

Note personali: la registrazione della data provenance viene effettuata in sequenza e immediatamente dopo l’esecuzione dell’operatore di cui si vogliono registrare gli effetti. Questo significa che se si verifica un errore durante il preprocessing dei dati definito da quello specifico operatore con interruzione della pipeline, non verrà registrata la causa e questo richiederà un debug manuale di quello step di preprocessing.

**Operazioni singole**

Value transformation

Salvataggio su Db dei file entities, activities e relations tramite lo script

create\_mongodb.py <dbname> <resourcesFolder>

es. python create\_mongodb.py census results/census\_prov

Lettura del contenuto del db tramite shell o MongoDB Compass

connectTo: mongodb://localhost/census

<la porta di connessione, su cui il db server è in ascolto di default è la 27017>

Pre-processing methods

ValueTransformation\_Prov.ipynb

Alla fine, tramite save\_all\_graph viene creato un documento di provenienza che può essere salvato come immagine png tramite il metodo prov\_to\_dot.

Questo produce una rappresentazione grafica dei dati di provenienza registrati che mostra come le varie entità sono state modificate. In questo caso ho preso in esame le prime 3 righe del dataset adult.data e convertito i valori “Bachelor” della colonna education in Nan.

a = df['education'][0]

df = df.replace(a, np.nan)

Ovviamente questo è un esempio giocattolo; in uno scenario reale value transformation può essere applicato per:

* sostituire i valori nan con degli 0

df.fillna(0)

* registrare i dati di provenienza

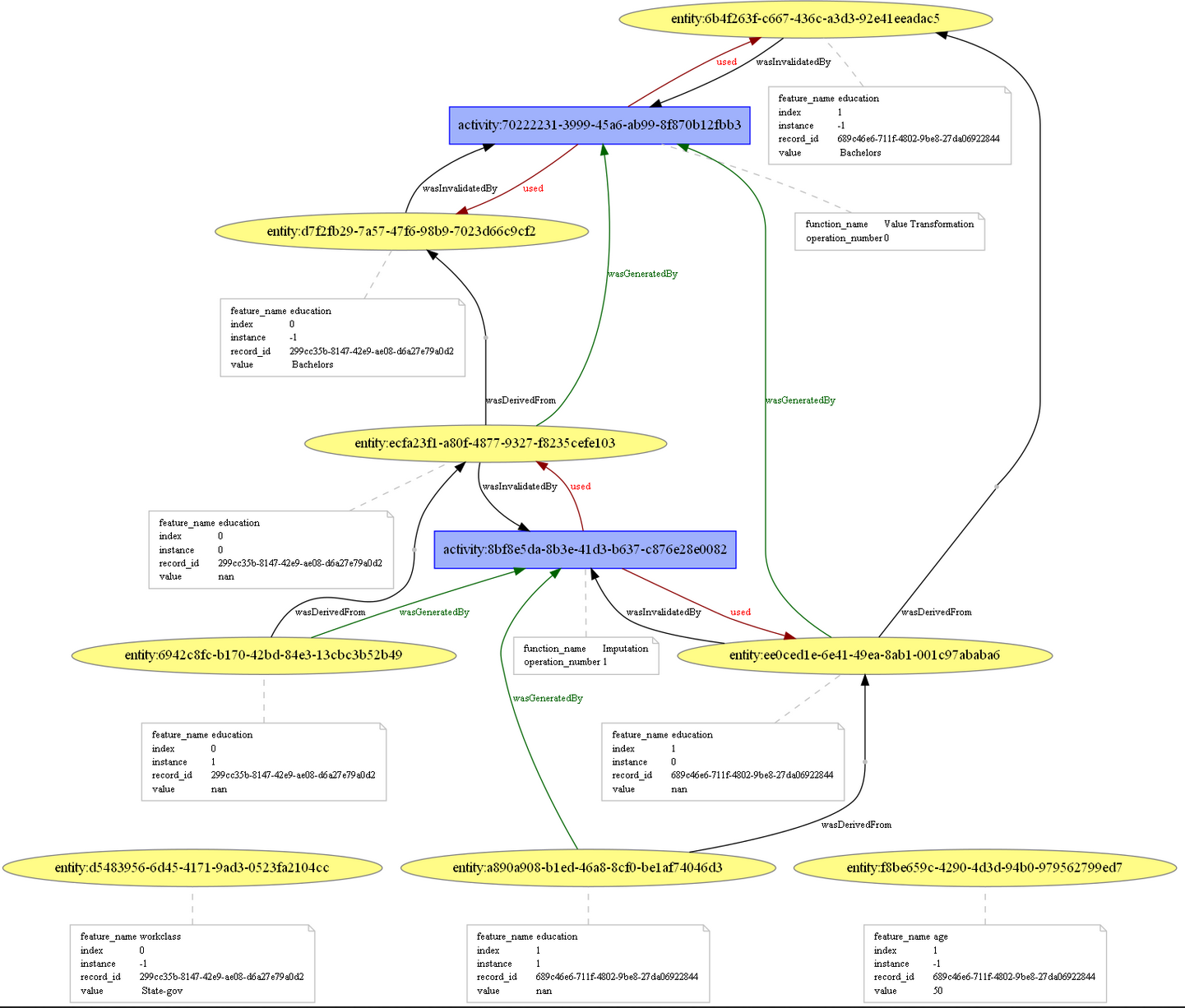
d = p.get\_prov\_value\_transformation(df,a)

* salvare e mostrare i dati sotto forma di immagine

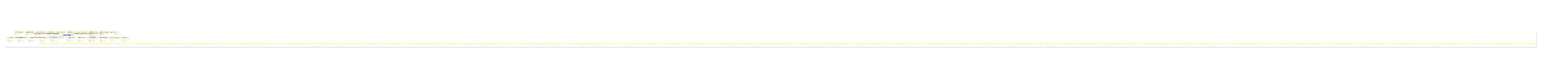
namefile = 'ValueImputation\_prov'

p.save\_all\_graph(namefile)

Image(savepath + namefile + '.png')



Bisogna far notare che in scenari reali il salvataggio dell’immagine di provenienza è sconsigliato perché finisce per produrre immagini schiacchiate e indecifrabili, come nel seguente esempio in cui ho voluto tracciare la trasformazione del valore della colonna education che ha coinvolto 50 record e ha prodotto la seguente immagine



Nonostante io abbia provato a fare lo zoom dell’immagine, questo è ciò che si riesce a vedere



**MongoDB**

per connettersi a mongoDB si utilizza la libreria python pymongo e bisogna avere un’istanza di MongoDB in esecuzione.

**Store data on MongoDB**

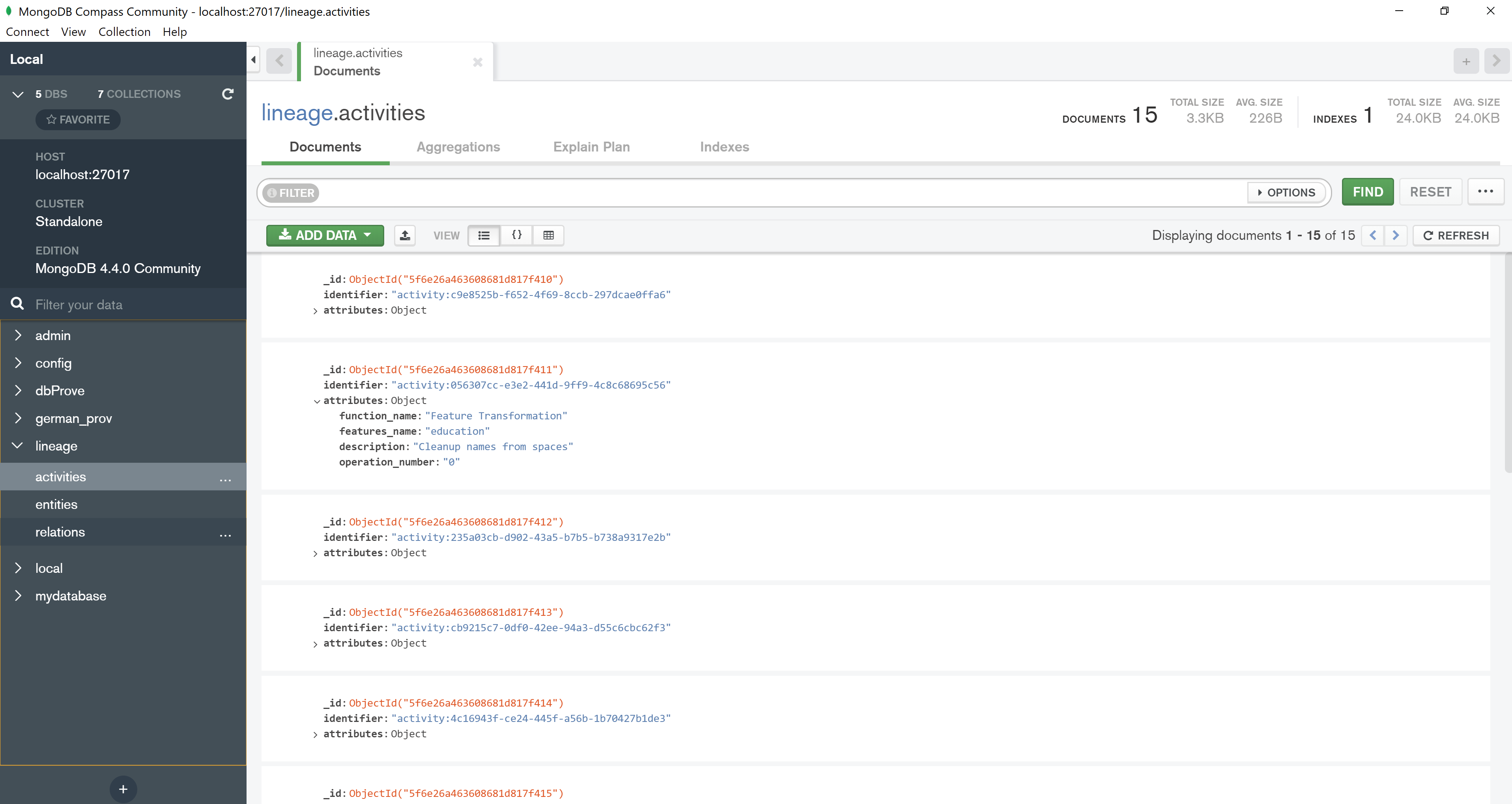
Per memorizzare i dati di provenienza catturati si ricorre allo script **create\_mongodb.py** alla root del progetto. Esso accetta come parametri:

* il nome del database in cui memorizzare i dati (entità, attività, relazioni)
* il path dove si trovano i file di provenienza salvati

Esempio d’uso:

* aprire una command line presso la root del progetto
* controllare che mongodb sia in esecuzione: mongod
* eseguire: python create\_mongodb.py lineage results\census\_prov

Con MongoDB Compass si ha una GUI tramite cui esplorare i dati salvati.



Per la memorizzazione su MongoDB vengono letti i file, il cui contenuto json è una lista di entità, attività o relazioni e vengono caricati tramite insert\_many nel db specificato.

**Queries**

Vengono qui discusse alcune implementazioni e risultati delle query che utente potrebbe voler fare a mongodb per indagare le informazioni di provenienza registrate.

Creare un db schema con nome german\_prov tramite lo script create\_mongodb.py, valorizzato con i dati di results/GermanCredit\_prov

python create\_mongodb.py german\_prov results\GermanCredit\_prov

Eseguire lo script

* python queries/dataset\_operation.py

Restituisce tutte le attività eseguite su GermanCredit\_prov con nome di attributo corrispondente a quello del feature name contenuto all’interno del file (checking è riportato nel file originale).  
Qui viene usata una semplice find sui nomi delle feature per il campo attributi nella collezione delle attività del db german\_prov.

activities.find({'attributes.features\_name': {'$regex': '.\*' + feature\_name + '\*.'}})

**Possibile output**

{'\_id': ObjectId('5f3ec2a80c589d159da775bb'),

'attributes': {'features\_name': 'checking',

'function\_name': 'Feature Transformation',

'operation\_number': '0'},

'identifier': 'activity:44f2148b-9267-43b4-ac6f-dc6a74fc79d6'}

{'\_id': ObjectId('5f3ec2a90c589d159da892f2'),

'attributes': {'features\_name': 'checking',

'function\_name': 'OneHot Encoding',

'operation\_number': '3'},

'identifier': 'activity:bf63830a-ed33-4afa-a7d5-f8b62fcc5901'}

mongo --port 27017: login al db sulla porta 27017

use <dbname>: seleziona lo schema <dbname>

show collections: mostra le tabelle

Se invece vogliamo vedere tutte le trasformazioni effettuate sulle feature

python queries\all\_transformations.py

all\_act = {}

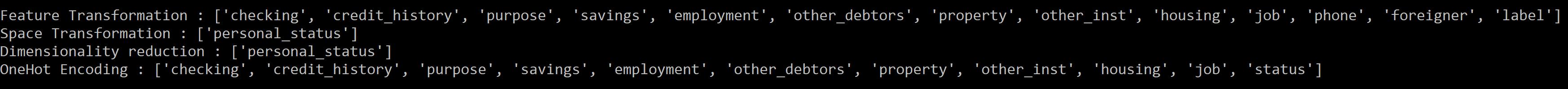
for act in activities.find():

function\_name = act['attributes']['function\_name']

all\_act.setdefault(function\_name,[]).append(act['attributes']['features\_name'])

Si tratta di usare una find e creare una mappa function\_name: features\_name.

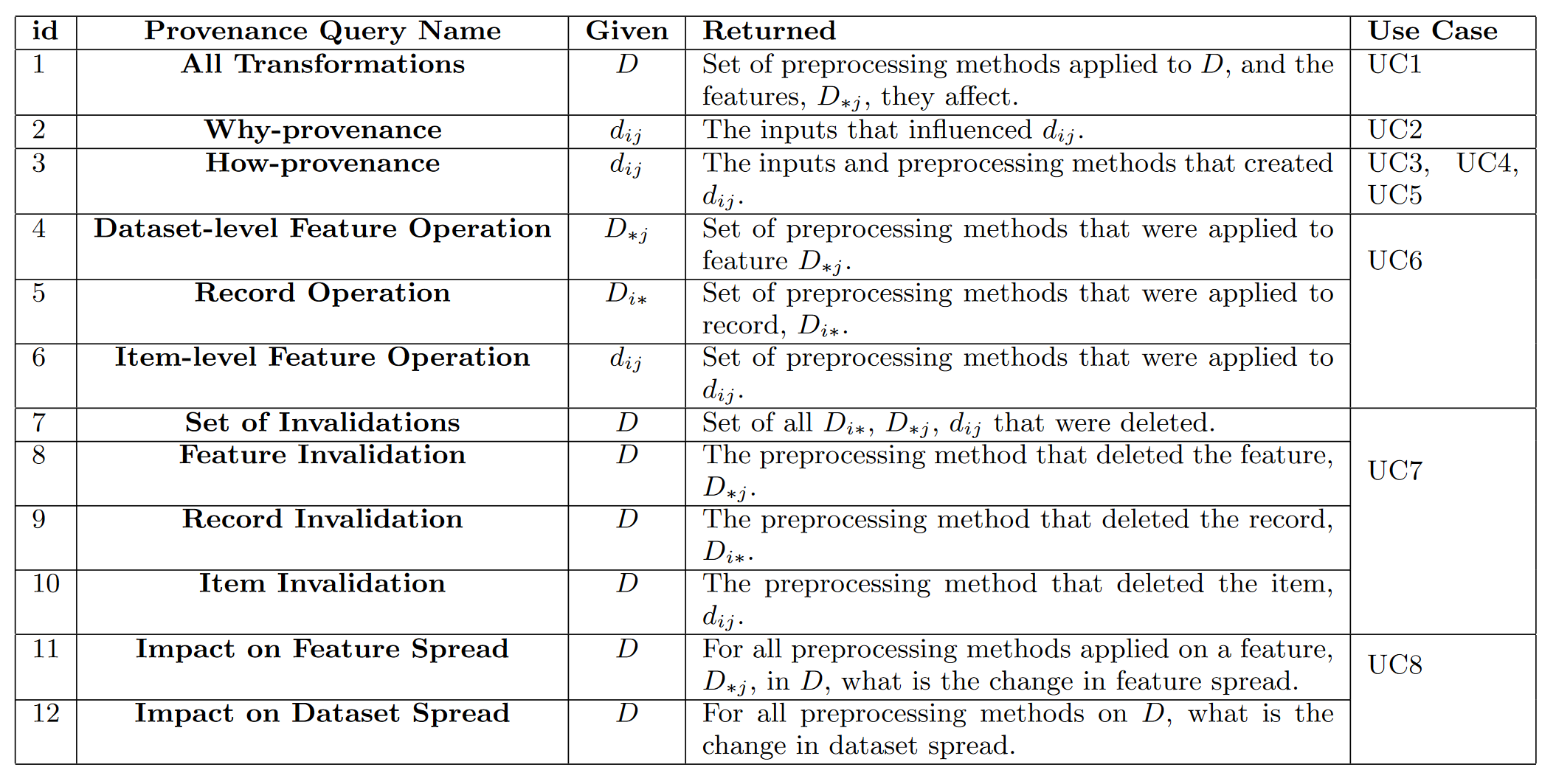
**Output atteso**



Differenze tra FP+ e FP, cosa fa FP+ per richiedere più spazio ma meno tempo? Suddivide la registrazione della provenance su più file.

**Neo4j**

Ho avuto anche l’occasione di provare Neo4j e memorizzare stavolta le informazioni in un Graph DB. Questo mi ha permesso di rispondere più efficientemente ad alcune query quando si deve navigare da un’entità ad un’attività tramite le relazioni che le collegano.



All transformations

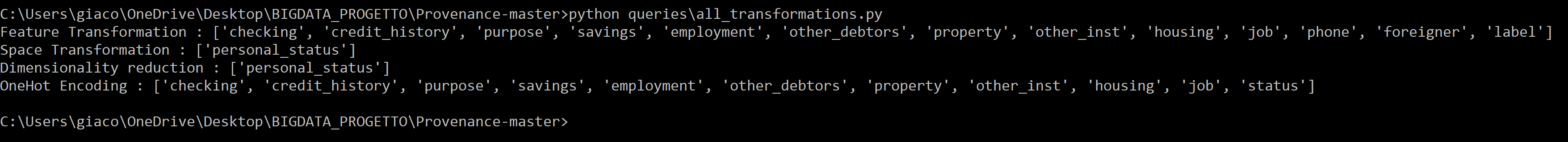
for act in activities.find():

function\_name = act['attributes']['function\_name']

all\_act.setdefault(function\_name,[]).append(act['attributes']['features\_name'])

for k, v in all\_act.items():

print(k, ":", v)



MATCH(a:Activity)

WITH a.function\_name as fn,collect(a.features\_name) as cll

UNWIND cll as nums

return fn,collect(distinct nums)



Se non avessimo disposto delle informazioni sulle feature modificate (entità) all’interno delle attività, grazie a Neo4j avremmo potuto fare:

MATCH(a:Activity)-[relatedTo]-(e:Entity)

WITH a.function\_name as fn,collect(e.feature\_name) as cll

UNWIND cll as nums

return fn,collect(distinct nums)

Record operation

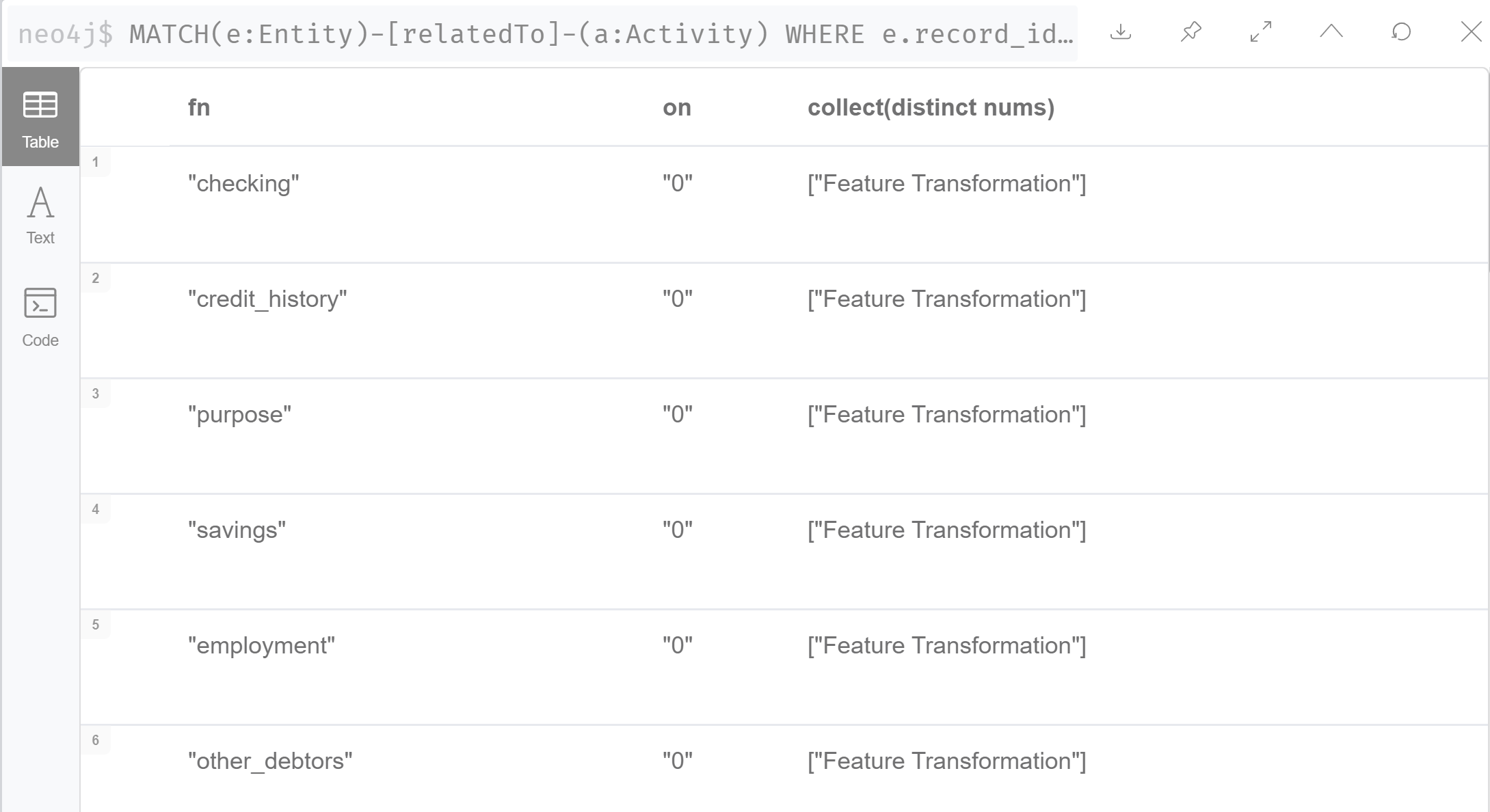
MATCH(e:Entity)-[relatedTo]-(a:Activity)

WHERE e.record\_id='f8c0771d-8147-4dc9-bd66-b6119755effe'

with a.features\_name as fn,a.operation\_number as on,collect(a.function\_name) as cll

UNWIND cll as nums

return fn,on,collect(distinct nums)



# ProvLake

### Informazioni

La complessità dei cicli di vita del ML dipende dall’enorme varietà di dati, attori in gioco, strumenti e workflow. Per potere ricreare un modello di ML dall’inizio o per poter spiegare alle parti interessate come è stato creato, diviene necessario tener traccia dei dati durante il suo ciclo di vita.

Il problema delle soluzioni già proposte è che non riescono ad affrontare la cattura e l’integrazione della provenienza con i dati di dominio e del ML utilizzati nell’ambito di numerosi workflow, mantenendo basso l’overhead dovuto alla cattura della provenienza.

Per questo motivo IBM ha creato nel 2017 una nuova rappresentazione dei dati di provenienza, PROV-ML, costruita utilizzando W3C PROV e lo schema ML.

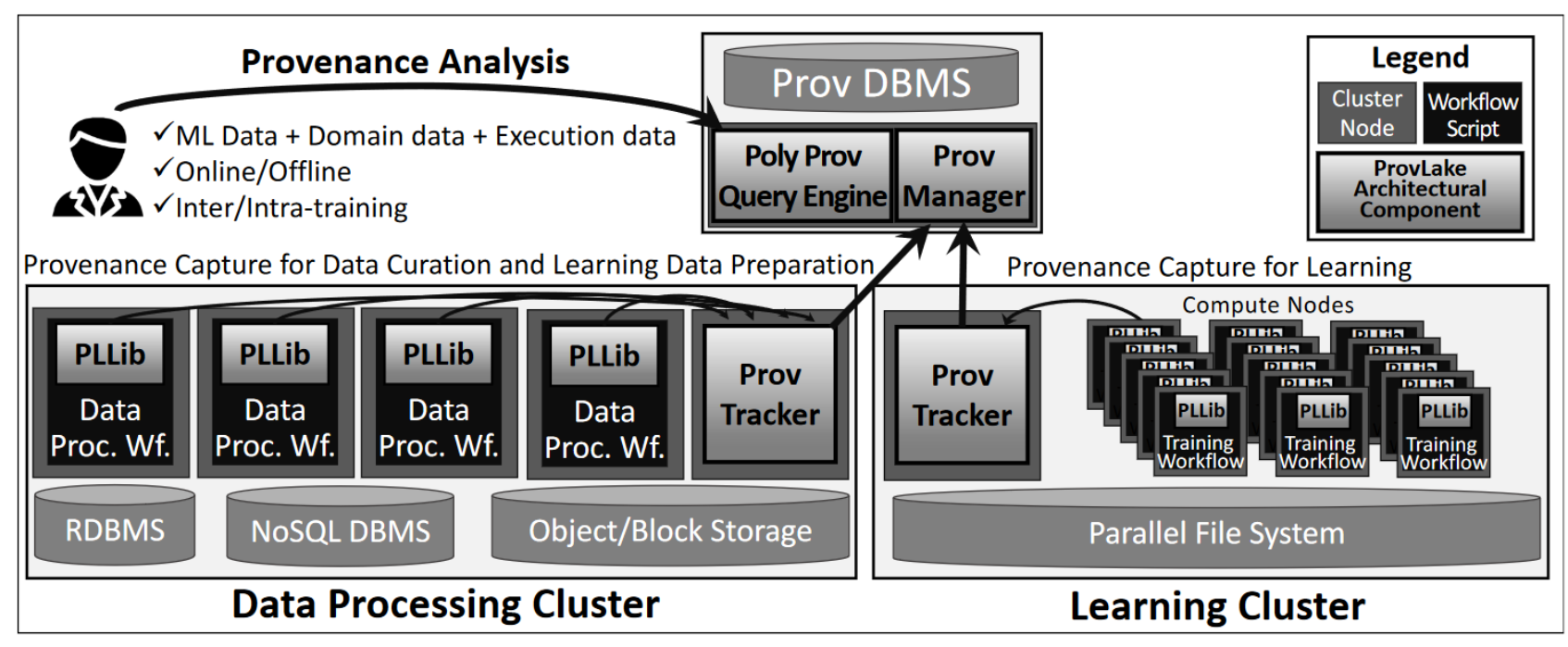
Attori in gioco:

* Scienziati di dominio
* Scienziati e ingegneri “computazionali”
* Scienziati e ingegneri del ML
* Specialisti di provenienza

Ciclo di vita del ML

* Preparazione/cura dei dati
* Preparazione dei dati per l’apprendimento
* Apprendimento

Architettura di ProvLake



I workflow sono instrumentati con PLLib, una libreria responsabile di catturare la provenienza negli script. Un task per la cattura della provenienza, delimitato da prov.in() e prov.out(), viene istanziato quando viene effettuata una trasformazione sui dati (chiamata di funzione, esecuzione di programma, iterazione). PLLib viene realizzata in modo da tenere basso l’overhead dell’esecuzione e da evitare di modificare troppo il codice utente. Per ridurre l’overhead vengono usati: accodamento della richieste, richieste asincrone, riduzione delle chiamate di sistema.

ProvTracker assegna degli identificatori unici ad ogni dato catturato da PLLib in modo da tracciare le relazioni tra i dati quando una trasformazione consuma i dati prodotti da un’altra trasformazione, popolando così il grafo dei dati. Se i valori dei dati sono riferimenti ad altri dati (es. file o identificatori in una tabella di un database) crea un arco tra il valore dei dati e il data store. Utilizza delle code di lavoro per raggruppare le richieste di provenienza prima di mandarle al ProvManager.

ProvManager è un servizio RESTful che riceve i dati di provenienza utilizzando il vocabolario PROV-ML e trasforma i dati in triplette RDF, inserendole in un bulk.

PolyProvQueryEngine fornisce le query per la provenienza, effettuate tramite endpoint utilizzando termini di PROV-ML. Se non è presente l’implementazione per un endpoint, gli utenti possono scrivere delle query grezze, sottometterle a PolyProvQueryEngine, che le invierà a sua volta al Prov DBMS.

Accodamento

#src/prov\_persister.py

if len(self.requests\_queue) >= self.bag\_size:

self.\_\_flush\_\_()

Per testare l’accodamento da script client.py

prov = ProvLake(online=True, should\_log\_to\_file=True,bag\_size=10)

in\_args = {"n": 5}

for x in range(10):

with ProvTask(prov, "factorial\_number", in\_args) as prov\_task:

factorial = calc\_factorial(in\_args.get("n"))

out\_args = {"factorial": factorial}

prov\_task.output(out\_args)

prov.close()

Per ogni provTask vengono inserite 2 richieste in coda, al riempimento della coda (di capacità massima 10 in questo caso) prov\_persister ne fa il flush che scrivere in questo caso su disco e invia la richiesta al servizio in ascolto, il quale poi potrà salvare in modo persistente i dati ricevuti su mongoDB.

Se prov.close viene richiamato prima del riempimento della coda provoca un flush immediato delle richieste in coda.

Dopo aver contattato uno degli autori, Renan Francisco Santos Souza mi ha indicato il repository git per la libreria provlake.

<https://github.com/IBM/multi-data-lineage-capture-py>

release: <https://github.com/IBM/multi-data-lineage-capture-py/releases/tag/0.0.72>

Purtroppo ProvLake server è un servizio proprietario IBM e non è quindi opensource. Il web service riceve i dati di provenienza catturati dalla libreria e si occupa di memorizzarli in un database che è ottimizzato per le query, in particolare query online (interattive) complesse che integrano diverse run di workflow.

IBM finora ha memorizzato i dati in Postgres e su DBMS Knowledge Graph data management systems, come AllegroGraph e BlazeGraph.

Obiettivi: MongoDB ha un ottimo throughput nell’inserimento e potrebbe essere utile per workload HPC. Anche delle semplici query possono beneficiare dal linguaggio di interrogazione di MongoDB. Verificare come query più complesse (join di diversi concetti) si comportano su MongoDB.

### Configurazione

Prerequisiti: python>=3.7, installare pip

* pip install flask pymongo requests-futures PyYAML
* clonare il progetto https://github.com/IBM/multi-data-lineage-capture-py
* unzip del progetto
* mkdir <nomecartella>
* cd <nomecartella>
* virtualenv .
* . ./bin/activate
* spostarsi sulla cartella multi-data-lineage-capture-py-master/src
* creare un file api.py e incollare il codice d'esempio (fornito sotto)

eseguire il codice tramite python <nomefile>

Questo lo script api.py, creato all’interno della cartella ./src

Se si opera in un'ambiente con un proxy aggiungere al costruttore del file provlake/prova\_persist.py:

self.session.trust\_env = False#ignora i proxy

#import flask

import os

import json

from flask import Flask,request

import pymongo

import logging

from random import randint

logger = logging.getLogger(\_\_name\_\_)

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)-8s %(message)s', datefmt='%a, %d %b %Y %H:%M:%S', filename='api.log', filemode='w')

logger.info("in esecuzione...")

app = Flask(\_\_name\_\_)

app.config["DEBUG"] = True

client = pymongo.MongoClient('mongodb://localhost:27017/')

mydb = client.mydatabase

print(client.list\_database\_names())

# usr = os.environ['MONGO\_DB\_USER']

# pwd = os.environ['MONGO\_DB\_PASS']

# client = pymongo.MongoClient("mongodb+srv://" + usr + ":" + pwd + "@firstcluster-obuqd.mongodb.net/test?retryWrites=true&w=majority")

# db = client['SampleDatabase']

# collection = db['SampleCollection']

@app.route('/', methods=['GET'])

def home():

return "<h1>prova</h1>"

@app.route('/', methods=['POST'])

def xx():

logger.info("ho ricevuto una richiesta post alla rotta /")

return "<h1>prova</h1>"

@app.route('/retrospective-provenance', methods=['POST'])

def xy():

logger.info("ho ricevuto una richiesta post alla rotta /retrospective-provenance")

req\_data = request.get\_json()

logger.info(req\_data)

record={"prov":req\_data}

mydb.provenanceFs.insert\_one(record)

return "ho ricevuto una richiesta post alla rotta /retrospective-provenance"

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Creare uno script client.py all’interno della cartella src.

Al suo interno un esempio giocattolo per la memorizzazione del workflow per il calcolo del fattoriale di un numero (n=5 in questo caso).

Script client.py

from provlake.prov\_lake import ProvLake

from provlake.prov\_task import ProvTask

def calc\_factorial(n):

num = n

result = 1

while num > 1:

result = result \* num

num = num - 1

return result

prov = ProvLake(online=True, should\_log\_to\_file=True)

in\_args = {"n": 5}

with ProvTask(prov, "factorial\_number", in\_args) as prov\_task:

out\_args = {"factorial": factorial}

prov\_task.output(out\_args)

prov.close()

Eseguire gli script:

Aprire 2 shell:

* sulla prima eseguire: python api.py
* sulla seconda eseguire: python client.py

Ci si aspetta la creazione di un file di log prov-wfexec\_1599919448.1285894.log contenente:

[{"prov\_obj": {"wf\_execution": "wfexec\_1599919448.1285894", "startTime": 1599919448.1285894}, "dataflow\_name": "NI", "act\_type": "workflow"}]

[{"prov\_obj": {"task": {"id": "0", "startTime": 1599919448.129513, "wf\_execution": "wfexec\_1599919448.1285894", "endTime": 1599919448.1298833, "status": "FINISHED"}, "dt": "factorial\_number", "type": "Input", "values": {"n": 5}}, "dataflow\_name": "NI", "act\_type": "task"}]

[{"prov\_obj": {"task": {"id": "0", "startTime": 1599919448.129513, "wf\_execution": "wfexec\_1599919448.1285894", "endTime": 1599919448.1298833, "status": "FINISHED"}, "dt": "factorial\_number", "type": "Output", "values": {"factorial": 120}}, "dataflow\_name": "NI", "act\_type": "task"}, {"prov\_obj": {"wf\_execution": "wfexec\_1599919448.1285894", "startTime": 1599919448.1285894, "endTime": 1599919448.1301029, "status": "FINISHED"}, "dataflow\_name": "NI", "act\_type": "workflow"}]

Nella chiamata al costruttore ProvLake abbiamo impostato l’attributo Online=True

Di default (senza cambiare il codice), per poter inviare i dati di provenienza ad un servizio

che poi li memorizzerà, è necessario:

* avere un servizio in esecuzione in localhost:5000
* avere 2 endopoint prospective-provenance e retrospective-provenance

Il servizio poi dovrà essere realizzato per occuparsi della memorizzazione persistente effettiva.

I dettagli sono disponibili in multi-data-lineage-capture-py/src/provlake/prov\_persister.py

FuturesSession di requests\_futures.sessions viene utilizzata per l'invio dei dati di provenienza al servizio in ascolto tramite post.

Assicurarsi che i dati di input/output memorizzati per la provenienza siano nel formato corretto (io ho usato json) o si incorre in

AssertionError: If you are not informing the prospective provenance, you must use dictionaries to be captured. Check your input or output arguments.

Inoltre se non si ha un servizio in ascolto su locahost:5000 (configurazioni di default, ma possono essere cambiate specificando un service\_url quando invochiamo il costuttore ProvLake, es: prov = ProvLake(online=False, should\_log\_to\_file=True, service\_url= "http://localhost:8080"), si incorre in:

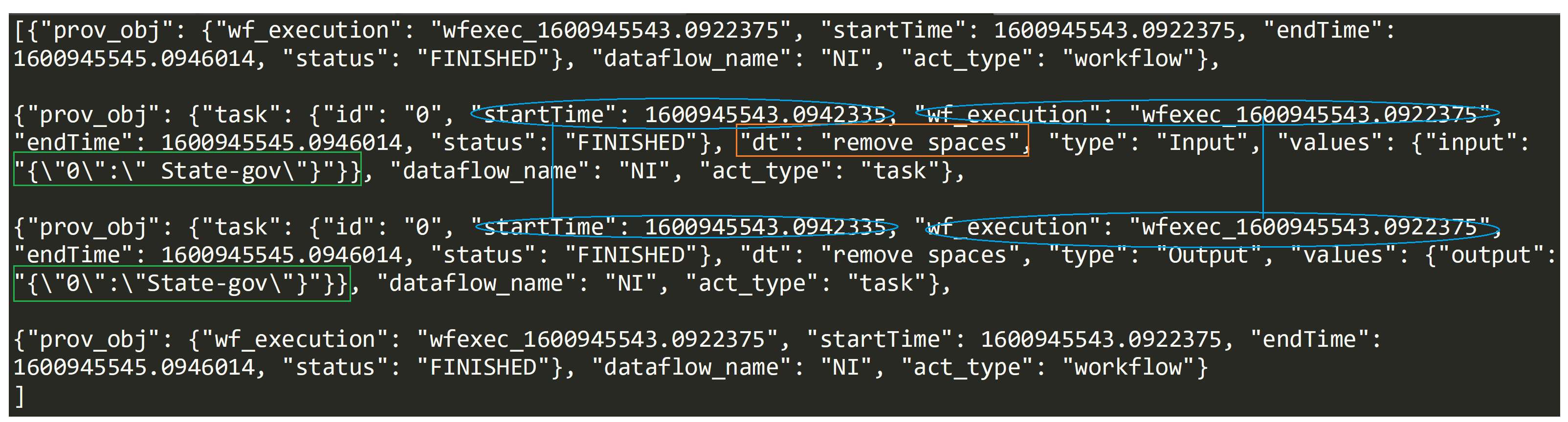
[Prov][ConnectionError] There is a communication error between client and server -> HTTPConnectionPool(host='localhost', port=5000): Max retries exceeded with url: /retrospective-provenance?with\_validation=False (Caused by NewConnectionError('<urllib3.connection.HTTPConnection object at 0x7f24764457f0>: Failed to establish a new connection: [Errno 111] Connection refused'))

Se invece riceviamo una risposta 502 Bad Gateway potrebbe dipendere dall’utilizzo di proxy che impediscono la comunicazione su localhost

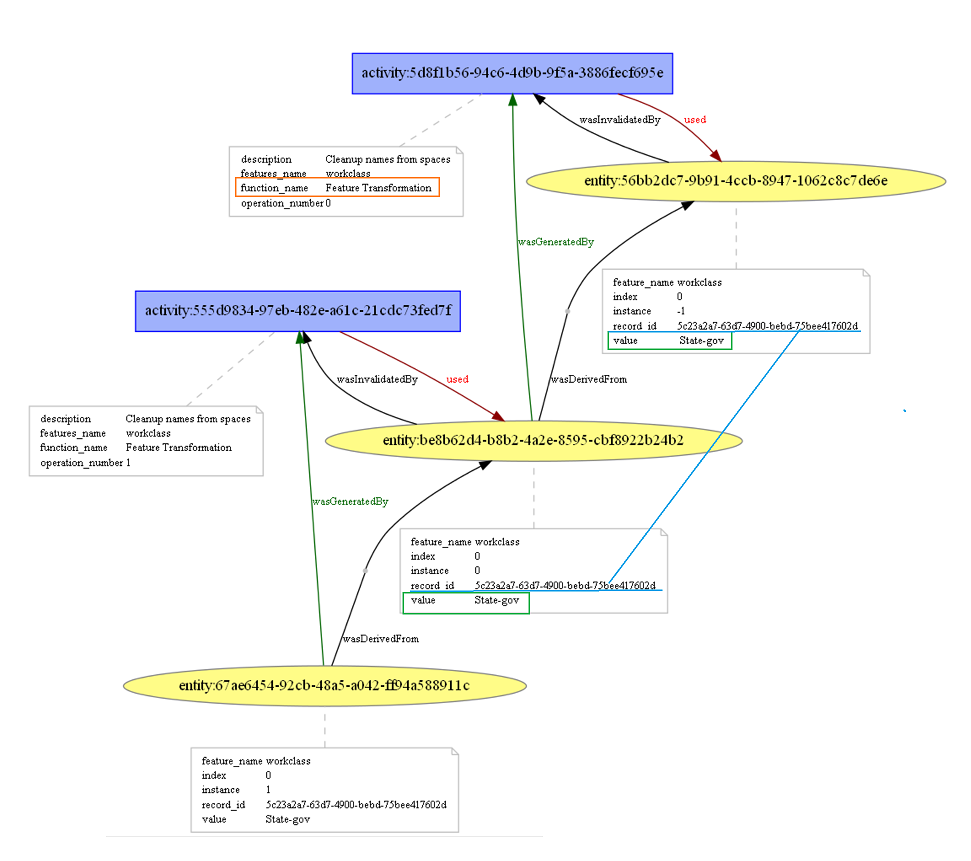
Confronto tra la data provenance di Giulia Simonelli (GS) e di IBM per il task di rimozione degli spazi dai valori delle colonne del dataframe adult.data

Per il confront prendo come esempio la modifica della colonna workclass del primo record

IBM

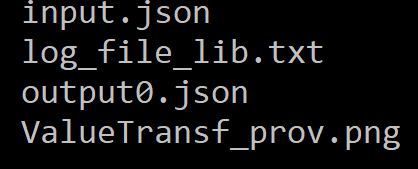


GS



IBM ha generato un unico file prov-wfexec\_1600944912.8674927.log contenente tutte le informazioni di provenienza tracciate.

GS ha generato una cartella con la seguente struttura



* input.json contiene un’entità per ogni feature del record in esame
* output0.json contiene le relazioni d’uso, invalidazione, generazione e derivazione per l’attività di rimozione dello spazio con riferimento all’entità coinvolta

Codice utilizzato per GS

Fare riferimento a CensusCleanup\_prov.ipynb, le modifiche sono indicate nei riquadri verdi dell’immagine a seguire



Codice utilizzato per IBM

#file IBM\_space\_removal.ipynb

# Cleanup names from spaces

col = ['workclass', 'education', 'marital-status', 'occupation',

'relationship', 'race', 'sex', 'native-country', 'label']

#col = ['workclass', 'sex', 'label']

prov = ProvLake(online=False, should\_log\_to\_file=True,bag\_size=5)

for c in col:

in\_args={"input": df[c].to\_json()}

print("input:\n"+df[c].to\_json())

with ProvTask(prov, "remove spaces", in\_args) as prov\_task:

df[c] = df[c].map(str.strip)

out\_args = {"output": df[c].to\_json()}

time.sleep(2)

prov\_task.output(out\_args)

print("\noutput:\n"+df[c].to\_json())

prov.close()

# Delta Lake e MLflow

### Configurazione

Requisiti: python >= 3.7, pip

* pip install mlflow
* eventualmente un account databricks community edition

Per gli esercizi eseguire il notebook Census\_withMLFlow.ipynb

### Informazioni + Esercizi

**MLflow**

È una piattaforma opensource per gestire il ciclo di vita del machine learning end-to-end. Offre 4 funzionalità:

* tener traccia degli esperimenti per registrare e confrontare i parametri e i risultati ([MLflow Tracking](https://mlflow.org/docs/latest/tracking.html#tracking))
* impacchettare il codice per il ML in una forma riusabile e riproducibile per condividere con altri data scientist o trasferire in produzione ([MLflow Projects](https://mlflow.org/docs/latest/projects.html#projects))
* gestire e fare il deploy di modelli per varie librerie per il ML su varie piattaforme ([MLflow Models](https://mlflow.org/docs/latest/models.html#models))
* fornire un punto di memorizzazione centrale per i modelli per gestire in modo collaborativo l’intero ciclo di vita di un MLflow Model, incluse le versioni del modelli e le annotazioni ([MLflow Model Registry](https://mlflow.org/docs/latest/model-registry.html#registry))

Il tutto viene offerto in modo agnostico rispetto alle librerie usate, ovvero è possibile utilizzarlo con qualunque librerie del machine learning e in qualunque linguaggio, dal momento che tutte le funzioni sono accessibili con un API REST e una CLI.

Il primo rilascio è avvenuto il 27 giugno 2018

Esiste anche un gruppo di supporto per fare domande. [https://groups.google.com/g/mlflow-users]

Ho utilizzato MLflow e Delta Lake su Databricks (compagnia fondata dagli autori di Apache Spark), creando un account gratuito (community edition).

L’account offre un cluster di un solo nodo (1 driver, 0 worker) con 15GB di memoria.

Src: https://github.com/databricks/tech-talks/tree/master/samples

Machine Learning Data Lineage with MLflow and Delta Lake Prep

Dataset: abitazioni a Boston (506 righe x 14 colonne)

[https://raw.githubusercontent.com/databricks/tech-talks/master/datasets/boston-housing.csv]

Alcune colonne interessanti:

I due notebook su cui si sono basati i miei test sono disponibili presso:

https://github.com/databricks/tech-talks/tree/master/samples

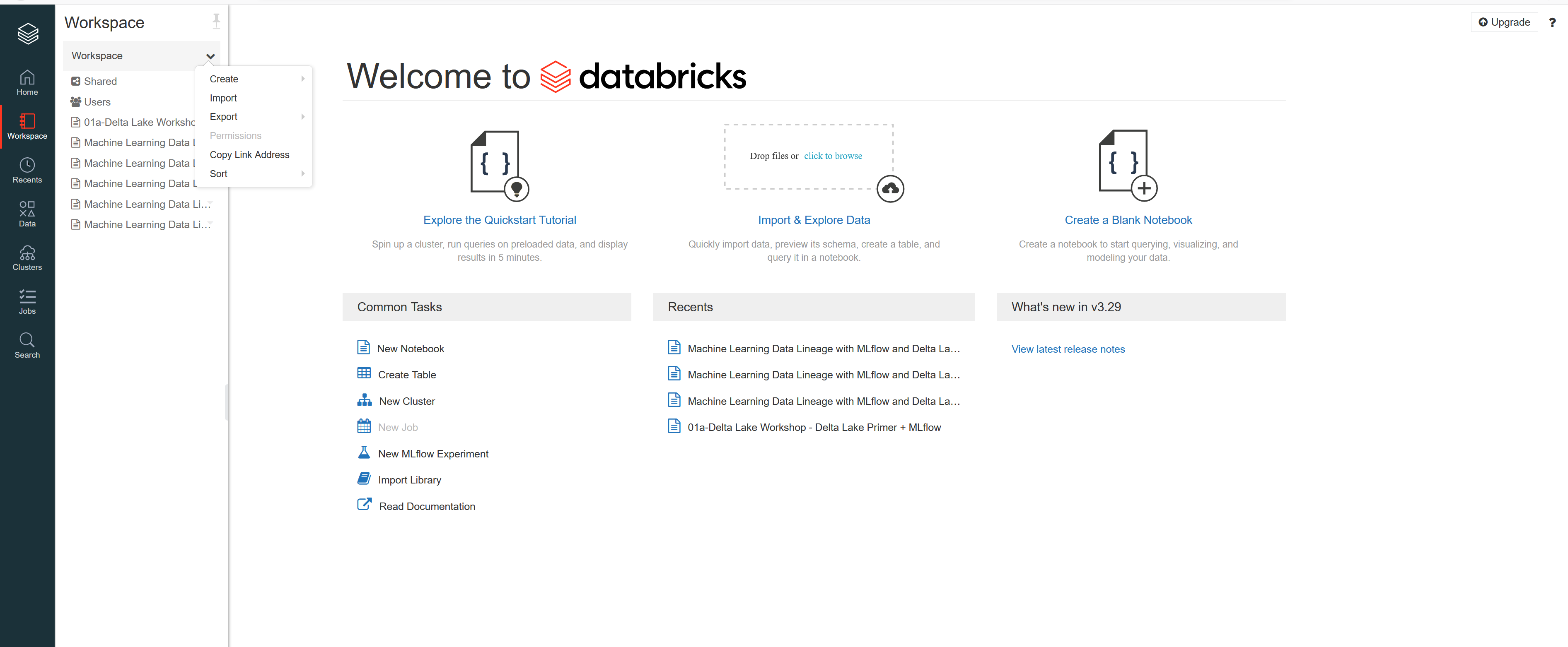
e sono:

* Machine Learning Data Lineage with MLflow and Delta Lake Prep.dbc per la preparazione dei dati
* Machine Learning Data Lineage with MLflow and Delta Lake.dbc per fare le analisi vere e proprie

Per utilizzarli è sufficiente farne il download e poi l’import su databricks dalla colonna workspace sulla sinistra partendo dalla home page

Alcune colonne interessanti :

* Crim: Tasso di criminalità a persona per città
* Zn: Parte del terreno residenziale suddivisa in lotti di oltre 25.000 m2
* Nox: Concentrazioni di ossidi di nitrogeno (ppm)
* rm: Numero medio di stanze per abitazione
* age: Proporzione di unità con proprietary costruite prima del 1940



Il notebook manca di alcuni comandi senza i quali non funziona nulla

dbutils.fs.cp("file:/dbfs/tmp/dennylee/samples/boston/boston-housing.csv", "dbfs:/tmp/dennylee/samples/boston/boston-housing.csv")

prima di

%fs ls /tmp/dennylee/samples/boston/boston-housing.csv

Perché: %sh mkdir d1 && wget … -O f1

Crea una cartella e un file sul filesystem locale (file:) e non sul file system distribuito (dbfs:)

CREATE DATABASE IF NOT EXISTS SAISNA\_2020;

Prima di

USE SAISNA\_2020;

dataframe: collezione distribuita di dati strutturati (simil tabella con schema in un database).

Per prima cosa vengono importate le librerie necessarie e fatto il download del dataset in formato csv.

La sessione di spark è l’entry point per il programma. Una chiamata read sulla sessione restituisce un dataframe reader.

Viene importato il dataset in un dataframe e gli diciamo di ignorare l’header e di inferire automaticamente lo schema. L’inferenza implicita viene fatta da spark che consulta alcuni valori per ogni colonna (data sample) e ne deduce il tipo di dato. In caso in cui gli diciamo di non inferire lo schema bisogna specificare esplicitamente lo schema.

Tramite createOrReplaceTempView viene creata una nuova vista valutata pigramente (lazily evaluated), che possiamo interrogare come una tabella del mondo relazionale.

Viene creato un database SAISNA\_2020 e una tabella boston\_housing\_delta su cui vengono caricati i dati.

**Data lake: repository centralizzato per i dati in grado di memorizzare dati strutturati (righe e colonne) e non strutturati (video, immagini, file binari, ecc.). I dati vengono salvati “così come sono”, senza dovervi subito imporre uno schema o una struttura.**

**Delta Lake**

[Delta Lake](https://delta.io) è uno storage layer open source che fornisce affidabilità ai data lake. Viene eseguito sopra data lake esistenti ed è compatibile con le API di Apache Spark.

Fornisce:

* Transazioni acide su Spark: i livelli di isolamento serializzabili assicurano che le letture non restituiscano mai dati inconsistenti
* Gestione di metadati che scala: sfrutta il potere di elaborazione distribuita di Spark per gestire tutti i metadati di tabelle fatte di molti file e con dimensioni dell’ordine dei petabyte
* Unifica streaming e batch: ingestione di dati di streaming, riempimento di dati storici per la parte batch, query interattivi sono pronti all’uso
* Rafforzamento dello scherma: gestisce variazioni dello schema in maniera automatica per evitare inserimenti di record problematici, di cattiva qualità durante l’ingestion
* Viaggio nel tempo: il versionamento dei dati abilita rollback, controllo di versioni storiche e permette di riprodurre gli esperimenti del machine learning
* Upsert e delete: sostiene operazioni di merge, update, delete per consentire casi d’uso complessi come cattura dei cambiamenti nei dati, operazioni slowly-changing-dimension (SCD), upsert nello streaming

Controllo di concorrenza

Fornisce garanzie di transazioni ACIDe tra le letture e le scritture. Ciò significa che:

* Per i sistemi di storage supportati (HDFS, Amazon S3, Microsoft Azure Storage), molteplici scritture su molti cluster possono modificare simultaneamente la stessa partizione di una tabella e vedere una vista istantanea della tabella, con un ordine seriale delle scritture
* I lettori continuano a vedere una vista istantanea consistente della tabella con cui il job Apache Spark ha iniziato, persino quando una tabelle viene modificata durante un job

Controllo di concorrenza ottimistico

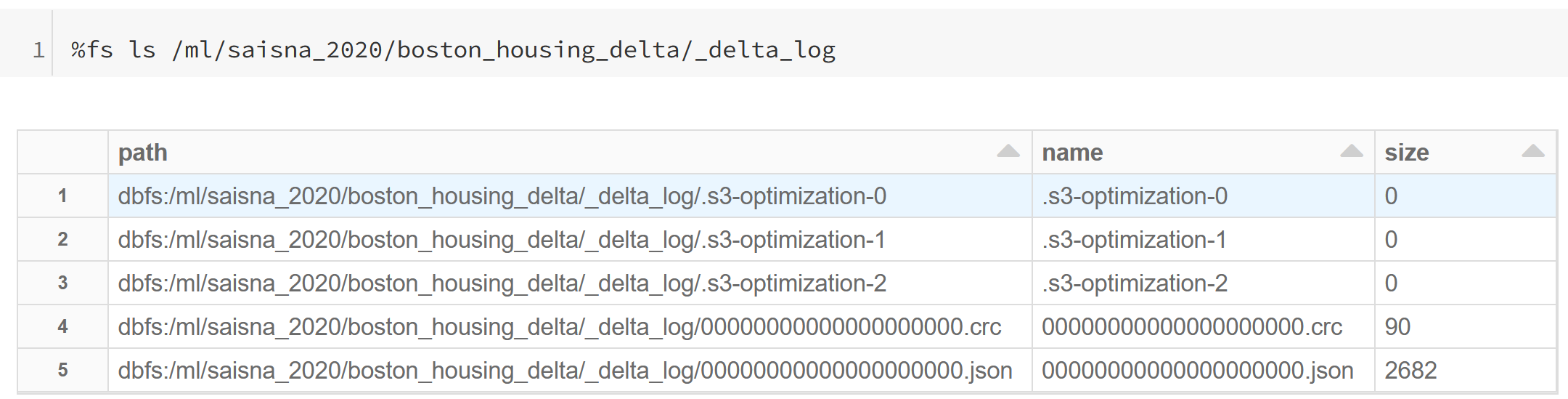
Delta Lake utilizza un controllo di concorrenza ottimistico per fornire garanzie transazionali tra le scritture. Secondo questo meccanismo, le scritture operano in 3 fasi:

Lettura: legge se necessario l’ultima versione disponibile della tabella per identificare quali file devono essere modificati (riscritti).

Scrittura: effettua tutti i cambiamenti scrivendo nuovi file contenenti i dati.

Validazione e commit: prima di fare il commit dei cambiamenti, controlla se i cambiamenti proposti vanno in conflitto con altri cambimenti che possono essere stati committati concorrentemente dalla lettura dell’istantanea. Se non ci sono conflitti, tutti i cambiamenti effettuati sono committati in una nuova istantanea con un numero di versione, e la scrittura termina con successo. Se ci sono conflitti invece l’operazione di scrittura fallisce con un’eccezione di modifica concorrente piuttosto che corrompere la tabella.

Nel nostro esempio non solo garantisce affidabilità, registrando i vari cambiamenti che vengono effettuati sulla tabella boston\_housing\_delta



ma fornisce anche prestazioni fino a 5-10 volte migliori rispetto alla creazione di una tabella standard che usa il formato Parquet.

Tramite tecniche quali **Data Skipping** e **ZORDER** è possibile saltare i dati non necessari quando si devono effettuare delle letture. È possibile specificare più colonne separate da virgola, con il rischio che l’efficacia però si riduca.

**Data Skipping** e **ZORDER Clustering** combinati permettono al Databricks Runtime di ridurre drammaticamente la quantità di dati che devono essere passati in rassegna per rispondere a query altamente selettive su tabelle Delta di grandi dimensioni.

Il data skipping interviene ogni volta che le query SQL o le operazioni su dataset includono filtri della forma “colonna operatore letterale”, dove colonna è un attributo (stringa, numerico, data, timestamp), operatore è un operatore di confronto (startswith, like, in), letterale è un valore o una lista di valori dello stesso tipo di quello della colonna.

La sua efficacia entra in gioco quando si hanno poche colonne su cui vengono applicati frequentemente dei filtri e che l’utente vuole accedere velocemente.

OPTIMIZE <table> [WHERE <partition\_filter>]

ZORDER BY (<column>[, …])

Le soluzioni degli RDBMS tradizionali (indici secondari) non sono pratiche nel contesto dei big data per ragioni di scalabilità.

E finora siamo stati abituati ad utilizzare il partizionamento (orizzontale).

Partizionamento: mantiene una sottocartella per ogni valore distinto della colonna di partizione. In questo modo le query sulla/e colonna/e di partizione possono beneficiare dal partition pruning, ovvero evitare di passare in rassegna una partizione che non soddisfa questi filtri.

**Le colonne su cui partizionare sono quelle che con maggiore probabilità saranno filtrare da query time-sensitive, che devono avere un rapido tempo di risposta.**

**Il problema è che si finisce per avere un enorme quantità di combinazioni uniche di valori, il che significa moltissime partizioni e file. Suddividere I dati su molti piccoli file porta vari problemi:**

**i metadati diventano tanto larghi quanto i dati stessi, con conseguenti problemi di performance nelle operazioni lato driver. Inoltre la compressione viene compromessa portando a sprechi di spazio e I/O più lenti.**

Perciò anche se il partizionamento dei dati in Spark in genera funziona benissimo per date o per colonne categoriche, non è adatto a colonne con alta cardinalità e in genere si limita a 1 o 2 colonne al massimo.

Data Skipping

Oltre al partition pruning, un’altra tecnica comunemente usata nel mondo delle data warehouse (che purtroppo manca in Spark) è l’I/O pruning. L’idea:

* tenere traccia di semplici statistiche come valori minimi e massimi ad un certo livello di granularità legato alla granularità dell’I/O
* sfruttare queste statistiche a tempo di esecuzione per evitare I/O non necessari

E questo è quello che fa il data skipping di Delta. Un nuovo dato viene inserito in una tabella Delta, vengono collezionate statistiche come minimi e massimi su file per tutte le colonne (include quelle annidate) dei tipi supportati. Quando poi viene fatta una ricerca su questa tabella, Delta prima consulta le statistiche per determinare quali file possono essere ignorati senza problemi.

Questa tecnica è ortogonale al partizionamento: funziona benissimo insieme ad esso, senza dipendervi. Ma l’approccio di indicizzazione probabilistico può dare dei falsi positivi, in particolare quando i dati non sono *clustered*.

ZORDER Clustering

Perché l’I/O pruning sia efficace i dati devono essere clustered in modo che gli intervalli min-max siano ristretti e idealmente non si sovrappongano e che quindi per una data ricerca il numero di hit sugli intervalli min-max sia minimizzato (massimizzando lo skipping).

A volte i dati sono clustered per la loro natura: ID crescenti, colonne legate al tempo di inserimento (date, timestamp) o la chiave di partizionamento. In altre situazioni è sempre possibile applicare il clustering con un ordinamento esplicito o partizionando i dati in intervalli prima degli inserimenti.

Ma è problematico se il carico di lavoro consiste di predicati, ciascuno relativo ad una singola colonna, ugualmente frequenti e rilevanti applicati a (es. n=4) colonne differenti.

In queste situazioni l’ordinamento lineare/lessicografico su tutte le n colonne favorirà la prima indicata, facendo un clustering perfetto dei suoi valori, ma apporterà poco, se non nulla (dipende da quanti valori duplicati ci sono sulla prima colonna) sulle altre colonne.

Quindi è altamente probabile che non ci sarà alcun clustering sull’n-sima colonna e quindi niente data skipping a tempo di ricerca.

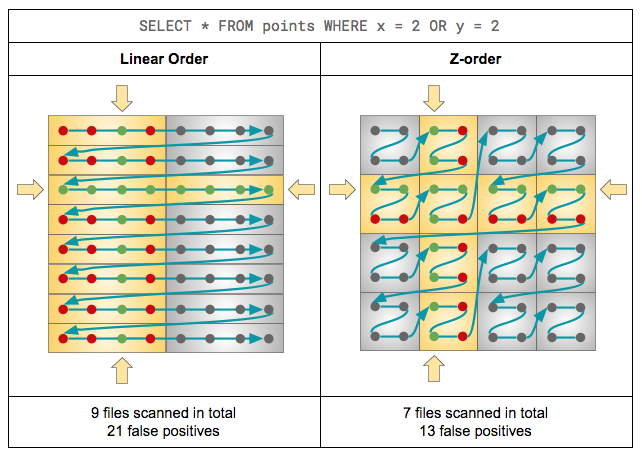
L’obiettivo è acquisire una simile efficacia lungo ogni singola dimensione.

Il problema diventa quello di cercare un modo per assegnare punti (dati) n-dimensionali su dei file, in modo che i punti assegnati allo stesso file siano vicini l’uno all’altro lungo ciascuna delle n dimensioni prese individualmente. Quindi mappare punti multi-dimensionali su valori mono-dimensionali per preservarne la località.

Questo problema viene affrontato nella computer grafica e nel geohashing. La soluzione consiste nelle curve space-filling che preservano la località (curve Z-order e di Hilbert).

Nella seguente immagine:

* i riquadri grigi sono dei file
* i punti grigi sono i dati
* i riquadri gialli sono i file letti
* i punti verdi indicano i dati che superano la condizione del filtro e che rispondono alla query
* i punti rossi sono i punti letti, ma che non soddisfano la condizione del filtro (falsi positivi)



Torniamo all’esempio. Ora che i dati sono stati salvati, possiamo iniziare con le elaborazioni.

Convertiamo il dataframe di spark, relativo ai dati caricati in tabella, in un dataframe pandas.

**Deciding Between Pandas and Spark**

Anche se i dataframe Pandas sono in memoria e single-server (la loro dimensione è limitata dalla memoria del server), permettono numerosi task quali: indicizzazione, ridenominazione, ordinamento, merging, aggiornamenti vari (aggiunte e rimozioni delle colonne), pulizia e preparazione dei dati con imputazione di valori mancanti o nulli.

Pandas è più flessibile, diretto, semplice da implementare di Spark, anche se non utilizza cluster e non scala.

Usando la funzione display possiamo visualizzare il dataframe in diversi modalità: tabellare, scatterplot, ecc.

Dallo scatterplot dell’esempio Boston Housing notiamo:

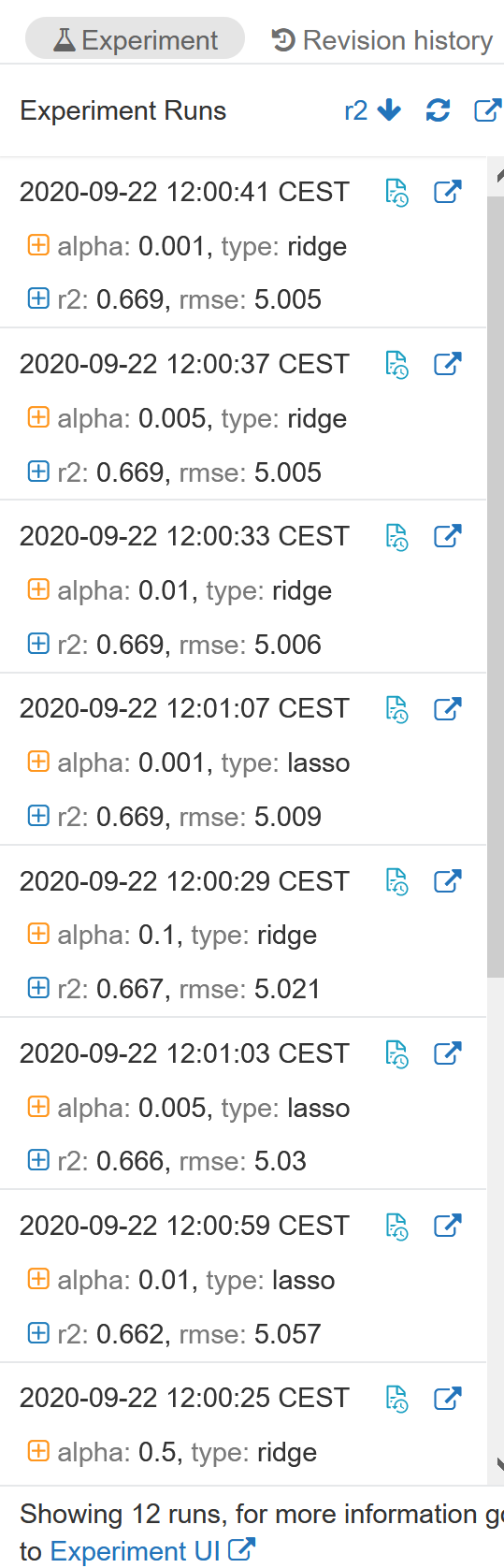
* una possibile correlazione positiva tra rm (media delle stanze per dimora) e medv (mediana delle abitazioni occupate da proprietari)
* una possibile correlazione negativa tra lstat (stato più basso della popolazione) e medv

Calcoliamo quindi la matrice di correlazione di tutti gli attributi con medv usando la funzione Pandas .corr

Rinomino le colonne e scarto le righe la cui mediana è nulla

Tramite scikit-learn addestro un modello di regressione lineare multipla e ne valuto le prestazioni misurando RMSE e R2, usando Ridge e Lasso.

Durante l’addestramento registro tramite mlflow una serie di informazioni: RMSE, R2, modello, valore di alpha e tipologia (Ridge/Lasso) scegliendo per alpha, di volta in volta, uno dei seguenti valori: 1.0, 0.5, 0.1, 0.01, 0.005, 0.001. Faccio una cross-validation.



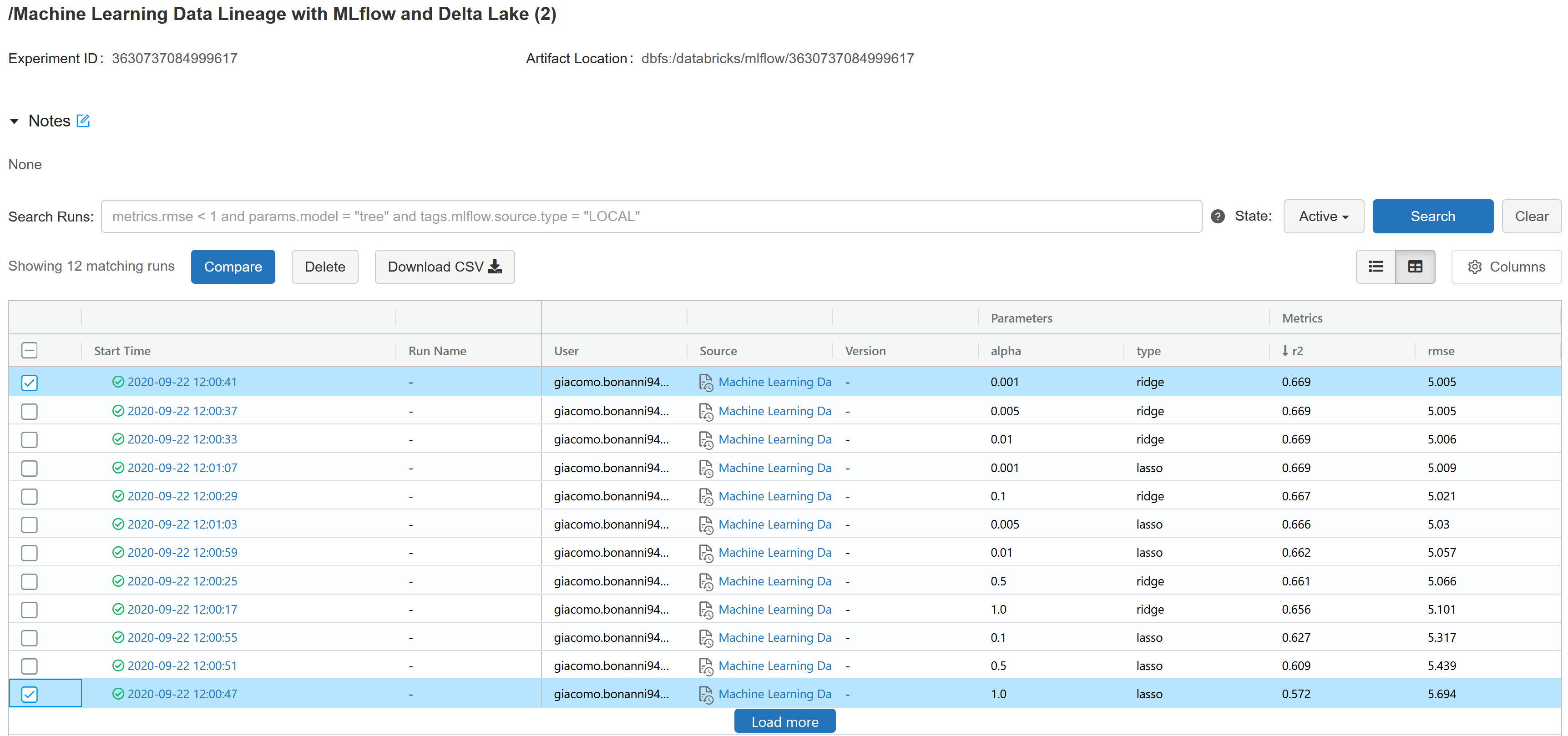
**R2**: calcola il coefficiente di determinazione. Rappresenta la proporzione di varianza di y (variabile da predire) spiegata dalle variabili indipendenti del modello. Fornisce un indicatore della bontà di adattamento e quindi una misura della probabilità che esempi non visti vengano predetti bene dal modello.

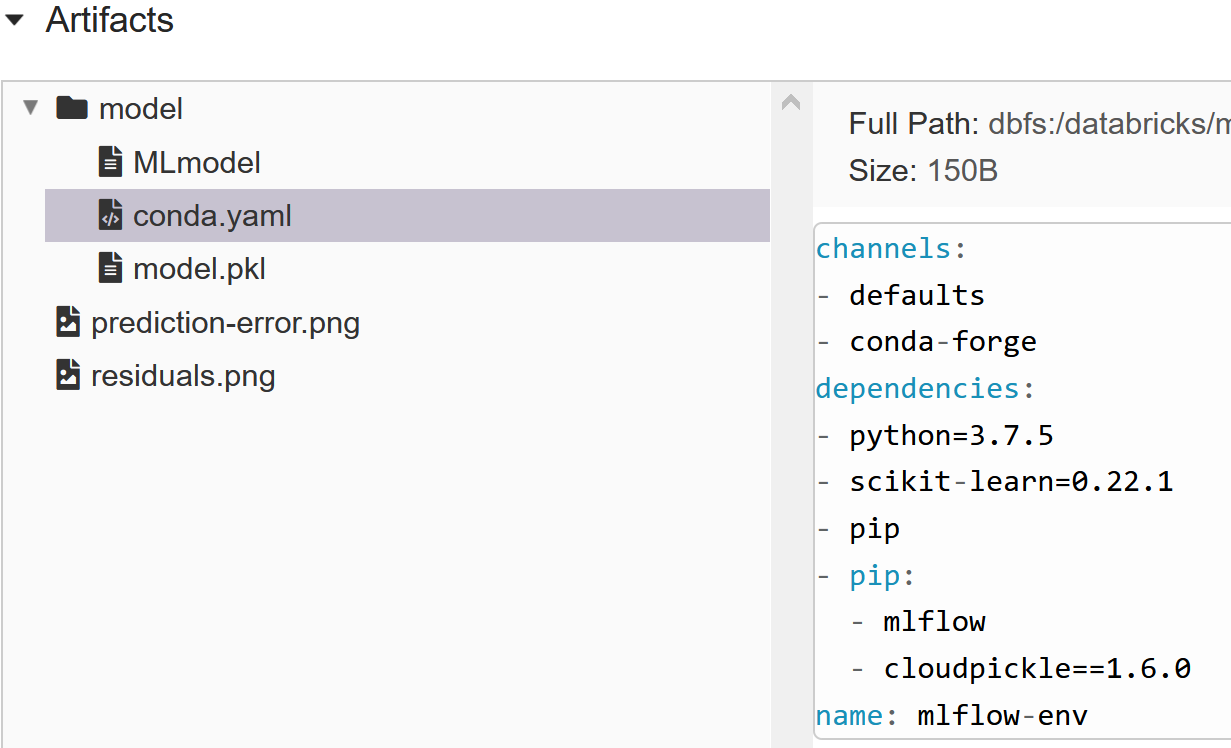
Il miglior score è 1.0 e può anche essere negativo (dipende dalla qualità del modello). Un modello costante che predice sempre il valore atteso di y, ignorando le feature di input, ottiene uno score R2 di 0.

**Mean squared error**: metrica relativa al valore atteso dell’errore al quadrato.

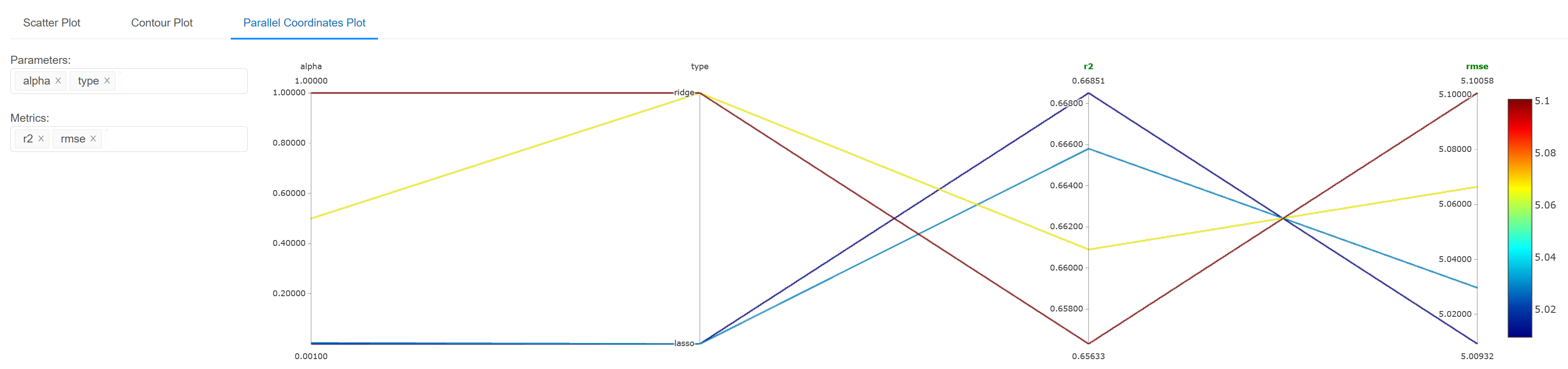
Sulla destra della pagina su cui stiamo facendo girare il codice è possibile selezionare la sezione “Experiment” per vedere i risultati dei nostri modelli, ordinarli in modo ascendente/discendente sulla base delle metriche o dei parametri (alpha, ridge, lasso) registrati.

Cliccando su View Experiment UI è possibile vedere i modelli in maggior dettaglio: parametri, metriche, librerie utilizzate





Con il confronto per coordinate parallele si ha una visione olistica dell’andamento dei modelli



# Altri test

Spiegazione del risultati per il dataset di consumi energetici

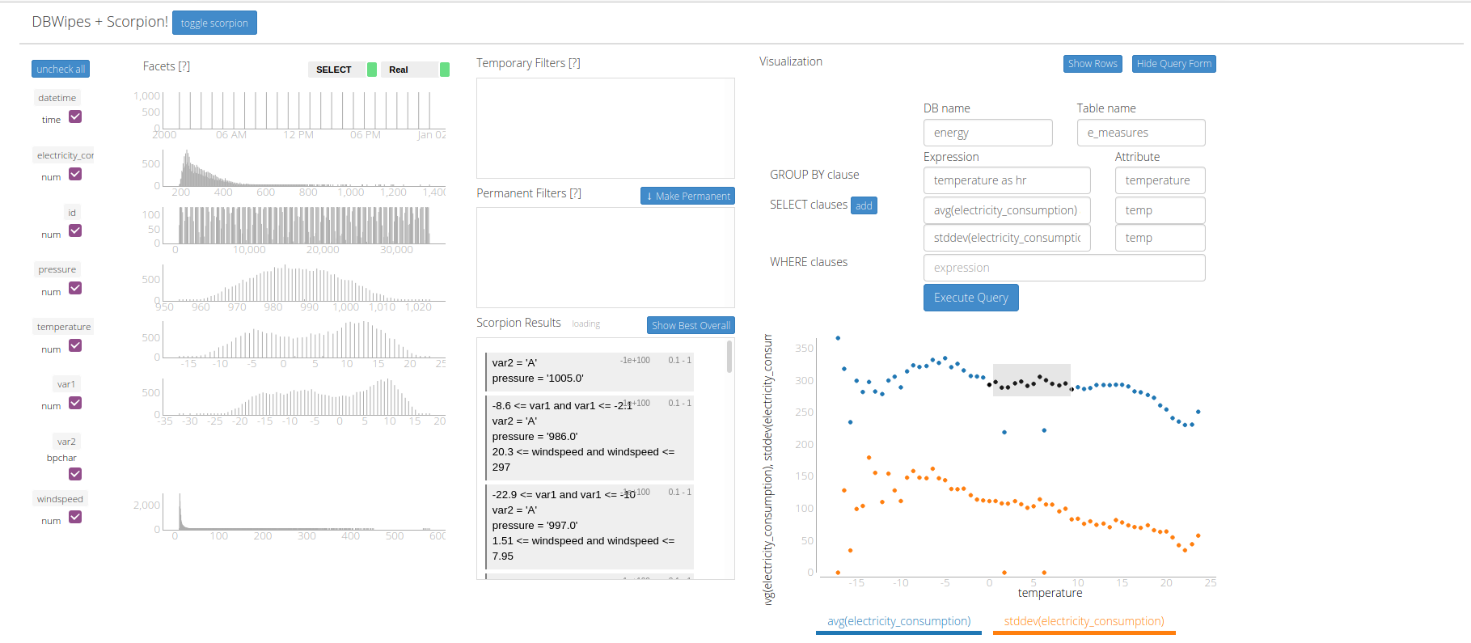
https://drive.google.com/file/d/1m8LZgZdb944afbc2Ze2boJtAgjFiW4mH/view?usp=sharing

tramite scorpion per la query che calcola il consumo elettrico medio dei vari sensori raggruppati per temperatura.

Qui come sempre

* createdb energy
* psql -f el\_data energy
* dbwipesserver.py --debug –threaded
* andare a localhost:8111

Notiamo che i 2 punti blu sottostanti la regione grigia selezionata vengono influenzati da variabili quali la pressione o la velocità del vento.



Scorpion riporta la dicitura “loading” perché non riesce a convergere durante la fase di merge. Questo è indicato dai log, che dopo 4 ore di lavoro continuano ad essere prodotti.

361 - DEBUG - merger noexpand pressure disc 31 options  
113 - WARNING -  \* Debugger is active!  
113 - INFO -  \* Debugger PIN: 291-201-739  
113 - INFO - 127.0.0.1 - - [24/Sep/2020 22:32:55] " [37mGET /api/status/?requestid=30 HTTP/1.1 [0m" 200 –

### Conclusioni

Durante la presente trattazione sono state affrontate diverse problematiche relative alla data provenance e a come questa venga registrata sia nel mondo dei big data che nel mondo relazionale.

Quando si opera per catturare la provenienza dei dati bisogna porsi prima di tutto delle domande: “quali query utente voglio soddisfare?”. Sulla base di queste domande vengono scelte caratteristiche quali la grana, il tipo di database in cui intendo registrarla (relazionale, document come MongoDB o persino un GraphDB come Neo4j).

Nei casi di studio visti:

* Scorpion ha cercato di fornire una spiegazione dei risultati per le query aggregative sui dati
* Giulia Simonelli ha cercato di rispondere a delle query sui dati di provenienza nella fase di pre-processing
* Souza (IBM Brasile) ha cercato di catturare tramite ProvLake le informazioni sugli input, gli output e le operazioni coinvolte registrando tutto all’interno di un unico file. La soluzione è altamente flessibile, ma quando aiuta poi questa decisione nella fase di interrogazione? Dipende da come verranno memorizzati questi dati.
* Mlflow ha cercato di registrare tutto e permette di avere una visione di insieme
* Delta Lake mediante il versionamento delle tabelle permette di mantenere uno storico delle operazioni svolte con l’obiettivo ambizioso di ottenere alta affidabilità. Ma tutti questi metadati occupano spazio e servono ragionamenti per cercare di ridurre al minimo la quantità di metadati necessari registrati, senza penalizzare la qualità informativa di questi dati

Il problema è ancora aperto. Quello che ho imparato da questa esperienza è che la tecnologia e le tecniche che si intende adottare dipendono sempre dal problema che è oggetto di studio, non tutte le tecnologie risolvono ogni problema.

Ringraziamenti

Ci tengo a ringraziare: Giulia Simonelli, autrice della libreria sulla provenienza nella fase di pre-processing; Matteo Interlandi e Gulzar, autori di Titian, e in particolare Gulzar con cui ho avuto occasione di parlare tramite Zoom e di scambiare email, sono stati davvero disponibili; Eugene Wu autore di Scorpion; Renan Souza autore di ProvLake.

### Bibliografia parziale

* [https://github.com/mlflow/mlflow/releases?after=v0.3.0]
* [https://cloud.google.com/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1]
* [https://www.mongodb.com/products/compass]
* [https://mlflow.org/docs/latest/index.html]
* [https://towardsdatascience.com/the-ultimate-tool-for-data-science-feature-factories-55aed0f70974]
* [https://towardsdatascience.com/feature-factories-pt-2-an-introduction-to-mlflow-873be3c66b66]
* [https://www.youtube.com/watch?v=hQaENo78za0]
* [https://www.learningjournal.guru/courses/spark/spark-foundation-training/spark-dataframe-basics/]
* **[https://databricks.com/discover/data-lakes/introduction]**
* [https://docs.databricks.com/\_static/notebooks/delta/optimize-scala.html]
* [https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html]
* [https://www.indellient.com/blog/a-journey-from-pandas-to-spark-data-frames]
* [https://scikit-learn.org/stable/modules/model\_evaluation.html]