Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Sede Di Bologna
Corso di Laurea Magistrale in Ingegneria Informatica

Predizione real-time da dati di sensori impiantistici e ambientali

Tesi di Laurea in

Data Mining

Relatore:
Chiar.mo Prof. Ing. Claudio Sartori

Candidato:
Marco Raminella

Correlatore:
Dott. Luca Paganelli

Anno Accademico 2018/2019

A mio papà, che crede in me
e a mia mamma, che avrei voluto tanto
vedere sorridere, dopo tutta la sua sofferenza

Indice

Introduzione **1**

1 Un problema di Data Science **2**

1.1 Il problema . . . 2
1.2 Il dataset . . . 3
1.2.1 Generazione dataset . . . 4
1.2.2 Modello previsionale . . . 7
1.3 Reti Neurali . . . 7
1.3.1 Reti Neurali Multistrato e Profonde . . . 9
1.3.2 Reti Neurali Ricorrenti . . . 11
1.3.3 Il modello NMT . . . 14

2 Dataset e training **20**

2.1 Ingestion . . . 21
2.1.1 InfluxDB . . . 22
2.1.2 Ingestion su InfluxDB . . . 23
2.2 Campionamento e aggregazione . . . 23
2.2.1 Campionamento . . . 25
2.2.2 Aggregazione e standardizzazione . . . 27
2.2.3 Data augmentation . . . 29
2.2.4 Training della rete . . . 30

3 Pipeline in batch e training **32**

3.1 PCollection . . . 33
3.2 Operazioni elementari . . . 33

3.2.1 ParDo . . . 33
3.2.2 GroupByKey . . . 34
3.2.3 CoGroupByKey . . . 35
3.3 Dataset shuffling . . . 36
3.3.1 Soluzione parallelizzata iniziale . . . 39
3.3.2 Soluzione parallelizzata scalabile . . . 42
3.4 Training della rete . . . 48
3.4.1 Modalità di training della rete . . . 49
3.4.2 Risultati con dataset estate 2018 . . . 50
3.4.3 Risultati con dataset pioggia 2018 . . . 53

4 Pipeline in streaming 61
4.1 Windowing . . . 61
4.1.1 Fixed . . . 62
4.1.2 Sliding . . . 63
4.1.3 Sessions . . . 64
4.2 Publish-Subscribe . . . 66
4.3 Watermark . . . 68
4.4 Triggering . . . 70
4.4.1 Comportamento del trigger . . . 71
4.4.2 Tipologie di trigger . . . 75

5 Sample and Hold in Streaming 86
5.1 Stateful ParDo . . . 89
5.1.1 Tipi di stato . . . 90
5.1.2 Timer . . . 92
5.1.3 Supporto Python . . . 94
5.2 Architettura . . . 95
5.2.1 Lettura da PubSub e filtraggio . . . 98
5.2.2 Windowing . . . 100
5.2.3 Trigger . . . 101
5.2.4 CombineByKey . . . 102
5.2.5 Stateful ParDo . . . 104

5.2.6 Scrittura su InfluxDB 111
5.3 Simulatore eventi 112
5.4 Testing e risultati 114
5.4.1 Prova con Completeness Trigger 114
5.4.2 Sliding windows e trigger combinati 115
5.4.3 Unaligned Delay Trigger 116
5.5 Modifiche e miglioramenti futuri 119

A Tecnologie utilizzate 123
A.1 Google Cloud Platform 123
A.1.1 BigQuery 123
A.1.2 Google Cloud Storage 124
A.1.3 Compute Engine 124
A.1.4 Google Dataflow 126
A.1.5 Piattaforma IA 127
A.2 Tensorflow e Tensorboard 128

Introduzione

L'utilizzo dell'Intelligenza Artificiale in ambito industriale sta prendendo piede negli ultimi anni e il caso studiato in questa tesi ne è la prova. Lo sviluppo della tecnologia ha reso disponibile sempre più potenza computazionale a minor prezzo, rendendo possibile l'utilizzo delle Reti Neurali Profonde, studiate fin dagli anni ottanta, in un modo che fino a non molti anni fa era economicamente insostenibile. Si andrà a vedere il caso concreto della realizzazione di un sistema che esegue previsioni in tempo reale su telemetrie di un impianto per la gestione delle acque, con lo scopo di assistere gli operatori nelle decisioni critiche da prendere in situazioni che potrebbero portare a un'emergenza. Sono state utilizzate tecniche allo stato dell'arte del Deep Learning per la realizzazione della rete previsionale, soluzioni di Big Data e Cloud Computing per la raffinazione dei dati grezzi e rendere possibile il training della rete neurale. Sono state studiate le basi teoriche richieste per realizzare un sistema in streaming, è stata poi progettata e realizzata una architettura apposita dedicata alla trasformazione in tempo reale dei dati per poter realizzare previsioni aggiornate.

Questo documento introdurrà il problema generale nel primo capitolo, nel secondo verranno introdotti più nel dettaglio i dati a disposizione e l'architettura previsionale. Sul terzo capitolo viene illustrato il metodo di trasformazione dei dati per renderli adatti al training della rete previsionale, per poi vedere i risultati dopo il suo addestramento. Nel quarto capitolo verranno introdotti i principi teorici essenziali richiesti per la trasformazione dei dati in tempo reale, con un esempio di riferimento applicato alle diverse tecniche disponibili. Nell'ultimo capitolo viene spiegato il caso d'uso rilevante affrontato dal candidato, andando nel dettaglio della progettazione e implementazione, i risultati ottenuti e i possibili cambiamenti futuri.

Capitolo 1

Un problema di Data Science

Questa tesi è risultato di un tirocinio presso il reparto di Ricerca e Sviluppo di Injenia Srl, azienda Premier Partner Google. Durante questo tirocinio è stato studiato uno dei progetti di Data Science seguiti dal team ed è stata progettata e realizzata una sua parte importante. Tale progetto consiste nella realizzazione di una PoC (Proof of Concept) funzionante di un sistema previsionale in tempo reale con utilizzo di tecniche Big Data e Machine Learning per un cliente di Injenia. Per motivi di riservatezza sono evitati riferimenti specifici al cliente e alla tipologia di impianto di cui ci si sta occupando.

1.1 Il problema

Il cliente dispone di un sistema centralizzato per gestire i propri impianti idrici, collegati al terreno, che vengono influenzati dalle condizioni meteo. Egli ha messo a disposizione le telemetrie dei sensori relativi a tali impianti e delle stazioni meteo della regione in cui sono allocati. Abbiamo a disposizione, inoltre, previsioni meteo sulle precipitazioni dell'intera regione in cui tali impianti sono allocati fisicamente. Un sottoinsieme dei segnali relativi a parti di tale impianto si rivela di vitale importanza per effettuare azioni concrete al fine di evitare lo scatenarsi di eventi potenzialmente dannosi, sia per gli impianti del cliente, sia per le infrastrutture che utilizzano tali impianti. L'obiettivo iniziale è la realizzazione di un PoC funzionante per prevedere l'andamento futuro dei segnali su un perimetro limitato degli impianti. Sono quindi state fatte scelte esplicite

al fine di dimensionare adeguatamente il sottoinsieme di dati presi in considerazione per limitare sia la dimensione del progetto, sia il perimetro preso in considerazione.

1.2 Il dataset

La soluzione ideata è caratterizzata dall'utilizzo di tecniche di Deep Learning applicato alle serie temporali per poter realizzare previsioni sui segnali richiesti, sfruttando al meglio i dati a disposizione. Una serie temporale è l'andamento di un fenomeno nel tempo. Considerato un solo fenomeno, la serie temporale che lo rappresenta in un certo intervallo temporale è caratterizzata da una serie di valori, ognuno con un proprio istante. Ogni valore è valido dall'istante che lo caratterizza fino a quello del valore successivo. Questo tipo di dato ha delle caratteristiche in comune al testo, alla musica, o anche alle sequenze di DNA [1]: si tratta di dati caratterizzati da sequenze, dove esiste una stretta correlazione fra ogni elemento consecutivo. In figura 1.1 vediamo l'esempio di una serie temporale.

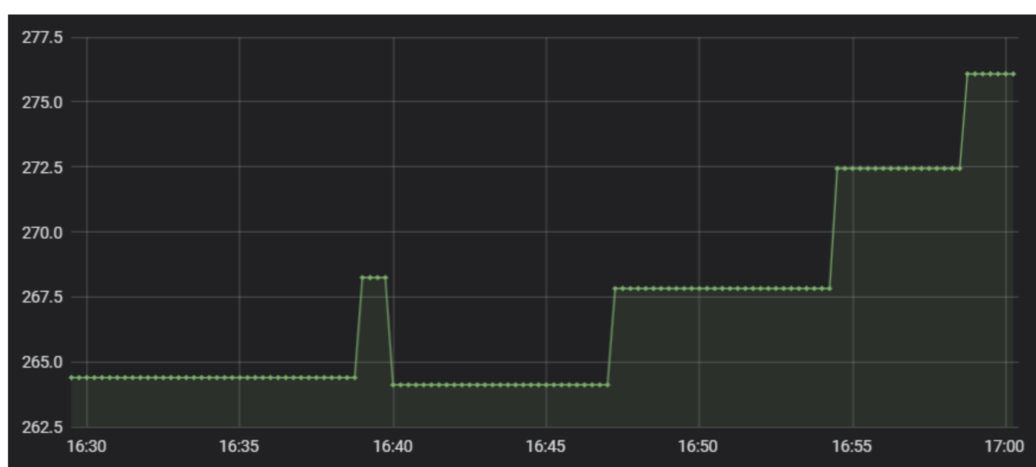


Figura 1.1: Una telemetria rappresentata come serie temporale in un intervallo dalle 16:30 alle 17 di una certa giornata. Per ogni puntino si ha un valore, in questa serie ci sono valori ad intervalli fissi di 15 secondi. Dalle 16:30 alle 16:38 circa ha valore 264.5, poi sale a 268 fino alle 16:39:45. Alle 16:40 si ha un nuovo valore di 264, che persiste nelle successive letture fino alle 16.47, dove la nuova lettura ha nuovamente valore 268. Prosegue così via salendo ogni qualche minuto, fino a ottenere alle 16:57 valore 276.

L'intera soluzione è stata ideata per poter realizzare tutte le operazioni di computazione

in maniera scalabile utilizzando le tecnologie allo stato dell'arte relative ai Big Data e al Machine Learning. L'obiettivo di questo capitolo è mostrare in maniera generale le parti principali dell'architettura e le loro funzioni.

1.2.1 Generazione dataset

In un progetto di Data Science, il compito iniziale consiste nell'analizzare i dati a disposizione e decidere come poterli utilizzare al meglio, al fine di realizzare un modello di Machine Learning che compia lo scopo richiesto. Tale modello avrà bisogno di dati strutturati in un certo modo per poter essere allenato e funzionare nel modo desiderato. In questo caso d'uso, abbiamo a disposizione due fonti di dati:

- SCADA
- Dati meteo

SCADA

Un sistema SCADA (Supervisory Control And Data Acquisition) è un sistema centralizzato di controllo che può gestire impianti e sistemi distribuiti in una vasta superficie territoriale. [2] Questo è possibile grazie alla suddivisione del sistema in:

- MTU: Master Terminal Unit
- RTU: Remote Terminal Unit

Le MTU caratterizzano il sistema centrale di controllo e sono connesse alle RTU. Le MTU contengono il software di interfacciamento a sensori ed attuatori e l'interfaccia grafica, che può essere distribuito e installato in sedi geograficamente distaccate. Le RTU forniscono informazioni sui dispositivi ad esse connessi e permettono di controllare, in remoto, eventuali dispositivi o attuatori. L'implementazione utilizzata dal cliente è in grado di fornire altissima scalabilità e granularità nella gestione dei sistemi. I dati dei sensori provenienti da SCADA sono caratterizzati da tre elementi principali:

- Tag
- Valore
- Timestamp

Il *Tag* identifica univocamente il sensore, che invierà una lettura ogni qualvolta vi è una variazione rispetto al valore precedente. Questa lettura avrà un proprio *Timestamp*, che indica l'istante di tale lettura, e un certo valore. Nello SCADA del cliente sono presenti le telemetrie relative a 11311 sensori. Questi sono collegati a diversi tipi di impianto, localizzati in regioni a remota distanza. La selezione dei segnali da utilizzare nel sistema previsionale è stata fatta sulla base dell'analisi dei processi operativi preesistenti del cliente, ritenuto esperto della fonte delle informazioni. Egli ha determinato un sottoinsieme dei sensori provenienti da SCADA relativi al perimetro degli impianti di interesse, raccogliendo tutti quelli necessari al fine di realizzare la PoC per gli impianti presi in considerazione. Si tratta in particolare di 51 sensori costituiti da:

- Tutti i sensori per la direzione e velocità del vento della regione dove è collocato l'impianto
- Livelli delle vasche
- Livelli dei condotti interconnessi
- Stato degli attuatori (pompe)
- Livello di apertura delle paratoie di interconnessione.

I dati relativi al meteo e alle stazioni metereologiche vengono tenuti tutti in considerazione, dato che sono stati ritenuti di vitale importanza. Nella seguente figura vediamo la schermata dello SCADA relativa alle stazioni meteo del cliente. Per ogni stazione meteo, si ha l'intensità di precipitazione di pioggia. Eventualmente, si hanno anche i dati di un sensore di intensità e direzione del vento.

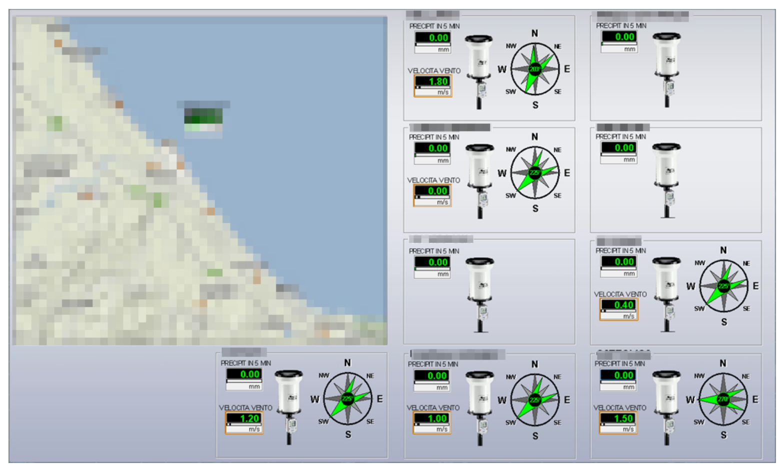


Figura 1.2: Schermata dello SCADA con i sensori direzione e velocità vento della regione (watermark per NDA).

Nella seguente figura vediamo la schermata dello SCADA che rappresenta uno degli impianti presi in considerazione. Sono stati considerati tutti i segnali relativi ai livelli idrici, per quanto riguarda le pompe non sono stati considerati i valori non ritenuti di rilevanza, come il consumo o il voltaggio.

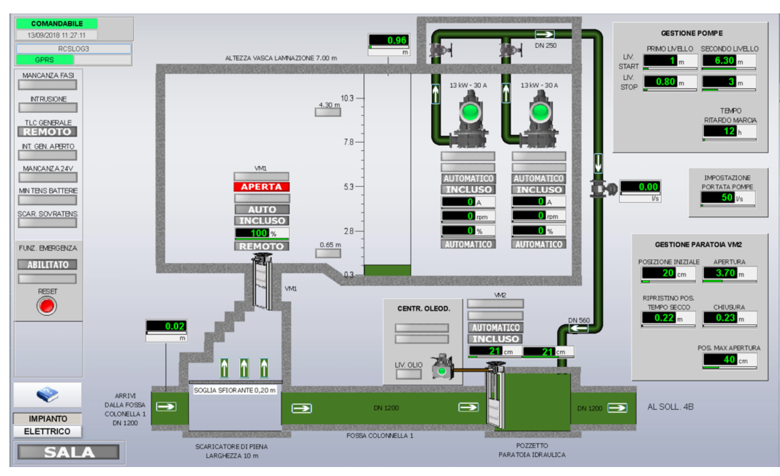


Figura 1.3: Schermata del sistema SCADA di uno degli impianti.

Dati Meteo

I dati meteo mettono a disposizione del cliente previsioni sulle precipitazioni distribuite su una griglia di 450 punti equidistanti. Le previsioni sono disponibili a 20, 40 e 60 minuti. Quelle a 20 minuti vengono trascurate perché sono fornite con eccessivo ritardo. Nella seguente figura vediamo una rappresentazione di come i dati meteo sono localizzati in una regione presa in considerazione. Per ogni punto avremo a disposizione una previsione.

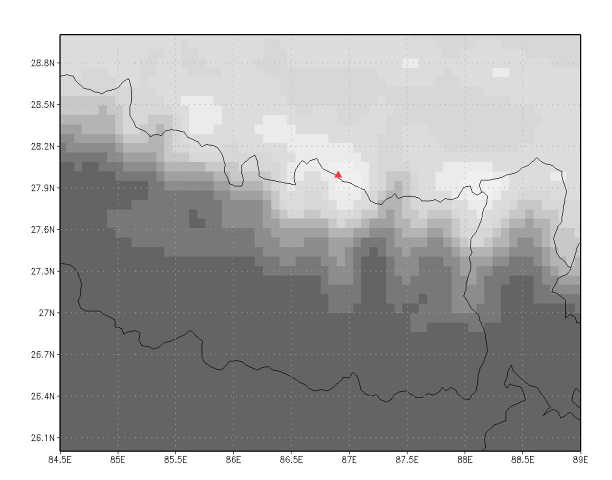


Figura 1.4: Esempio di matrice di punti meteo. Ogni punto, visto come un pixel in questa figura, avrà una certa quantità di precipitazione prevista: un pixel più scuro rappresenta un maggiore quantitativo.

Le telemetrie dei sensori e i dati meteo possono essere rappresentati come delle serie temporali.

1.2.2 Modello previsionale

È stato studiato un tipo di rete neurale realizzata per poter identificare e lavorare questo tipo di sequenze: i Recurrent Neural Networks, che vedremo nel seguito.

1.3 Reti Neurali

Le reti neurali artificiali sono una tecnica diffusa di Machine Learning che simula il modello condiviso fra i biologi per rappresentare i neuroni e il sistema nervoso che li

collega. La struttura elementare è il neurone, noto anche come Perceptron.

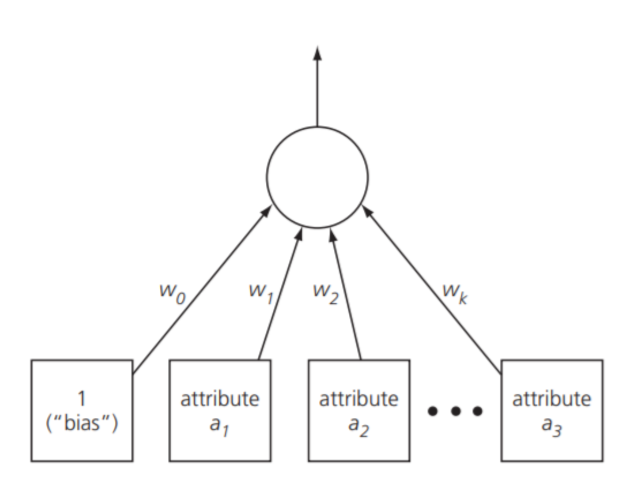


Figura 1.5: Un neurone o Perceptron [3].

Un Perceptron contiene uno strato di segnali di input e un nodo di output. Il nodo di output è una combinazione lineare degli input, moltiplicati per il loro peso w_i. L'uscita del Perceptron può essere matematicamente descritta con la seguente formula:

$$w_0 + w_1a_1 + w_2a_2 + ...w_ka_k = 0$$

Per poter funzionare, una rete neurale ha bisogno di una fase di apprendimento (o *learning*), durante la quale viene sottoposta a una insieme conosciuto di esempi per i quali è noto l'obiettivo richiesto. L'insieme conosciuto è detto dataset di *training.* L'addestramento (o *training*) viene eseguito sottoponendo la rete a tali esempi, andando a modificare i pesi delle connessioni fra i neuroni per ottenere l'output desiderato. Per testare i risultati ottenuti è opportuno verificare, tramite una qualche metrica, l'uscita della rete con un altro dataset di esempi noti, quest'ultimo noto come dataset di *evaluation.* La bontà della rete neurale sarà determinata dai risultati della evaluation: se la rete "funziona" correttamente con i dati di training, ma non con quelli di evaluation, significa che è andata in *overfitting*, dal momento che ha semplicemente memorizzato i dati in ingresso, piuttosto di ricavare una funzione più generale per realizzare lo scopo richiesto. Il perceptron che abbiamo visto sopra è in grado di apprendere tramite addestramento una funzione lineare che distingue oggetti di due classi differenti. Nella figura in seguito

vediamo un semplice esempio in cui abbiamo la classe “+” e la classe “-“. Quello che viene appreso dal perceptron è la linea che separa le regioni in cui queste due classi si trovano. Il nostro perceptron quindi, ricevendo un nuovo elemento “+”, darà la stessa uscita che ha appreso dagli altri esempi “+”.

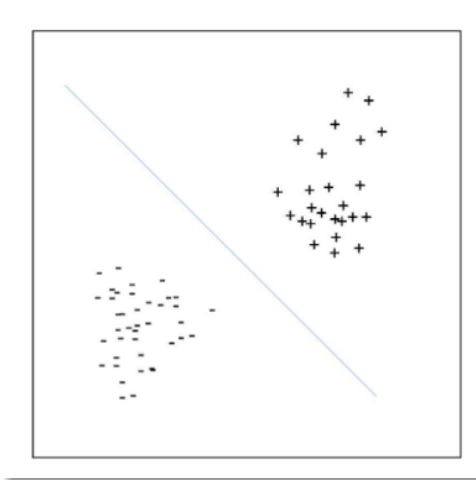


Figura 1.6: Esempio di un iperpiano con elementi separabili linearmente [3].

Le funzioni che, moltiplicate per il peso, collegano gli attributi agli output, dette *funzioni di attivazione*, sono tipicamente diverse dalla moltiplicazione lineare. Questo permette di ottenere separazioni anche fra regioni non separabili linearmente. Combinando più perceptron, realizziamo uno strato di rete neurale, che permette di avere più di una uscita. Questo permette di distinguere fra più di due tipi diversi di elementi, avendo una cardinalità maggiore di valori in uscita.

1.3.1 Reti Neurali Multistrato e Profonde

Per apprendere funzioni più complesse, si possono organizzare più perceptron in una struttura gerarchica, realizzando una rete multi-strato. Gli strati intermedi vengono chiamati *hidden layer*, in quanto non sono collegati direttamente all’ingresso o all’uscita. Se abbiamo più di un hidden layer si parla di Reti Neurali Profonde, note come *Deep Network*. Nella prossima figura vediamo un esempio di rete neurale multistrato. Abbiamo un primo strato di input, che non esegue alcuna computazione, dal momento che abbiamo 5 perceptron con solamente un ingresso e tre uscite. Al centro abbiamo due hidden

layer, costituiti da tre perceptron ciascuno. Tutti i perceptron sono collegati a tutti i segnali dello stato precedente, in questo caso si parla di uno strato *fully connected*, noto anche come *dense*. Ogni connessione avrà il proprio peso. Infine, in uscita, abbiamo un perceptron che combina le uscite del secondo strato hidden.

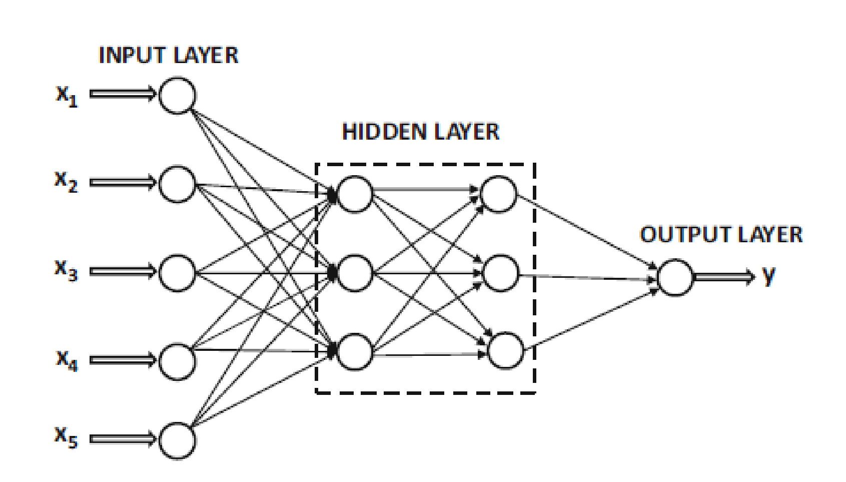


Figura 1.7: Esempio di una rete neurale multistrato [1].

Esistono moltissimi tipi di reti neurali multistrato, valide per i casi d'uso più disparati, con tipi di strati e caratteristiche ben più complessi di quelli mostrati fino a qui. Una caratteristica essenziale dei dati che vengono utilizzati in queste reti è l'atomicità, ovvero la separazione fra gli elementi. Questa caratteristica è valida, per esempio, se consideriamo diverse immagini e abbiamo bisogno di classificare l'oggetto raffigurato in ciascuna immagine. Nella prossima figura vediamo qualche esempio di classificazione di una rete neurale realizzata per il concorso annuale ImageNet, che dal 2010 rappresenta lo stato dell'arte nell'identificazione automatica del contenuto di immagini. Abbiamo una serie di immagini, per ognuna di esse viene identificato cosa vi è rappresentato.

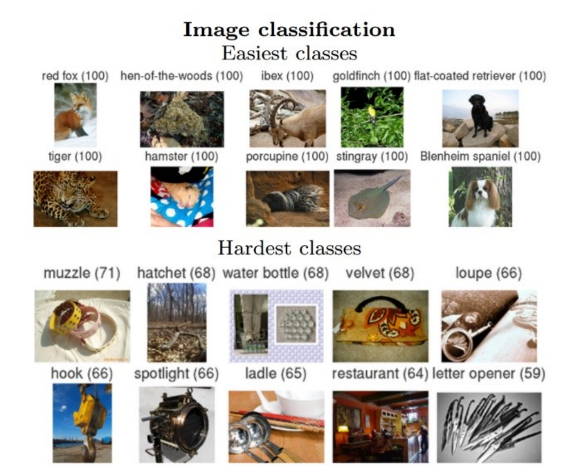


Figura 1.8: Alcune classificazioni di immagini della ImageNet challenge [4]. Ogni immagine è classificata da un Deep Network a un possibile oggetto o animale, con una affidabilità indicata fra parentesi.

Ognuna di queste immagini è a sé stante, e non vi è alcuna correlazione fra un'immagine e quella successiva. Di conseguenza la rete può considerare ogni ingresso completamente indipendente rispetto al successivo o al precedente. Questa caratteristica non vale per il dato in analisi nel nostro caso, che sono serie temporali.

1.3.2 Reti Neurali Ricorrenti

Per il Deep Learning sui dati caratterizzati da frasi di testo, serie temporali e altre sequenze discrete come il DNA, sono state ideate delle tipologie di reti note come Reti Neurali Ricorrenti. Questo tipo di reti permette di realizzare una interazione fra neuroni che ricevono gli elementi singoli di queste sequenze. Vediamo un esempio elementare di una rete di questo tipo. Supponiamo di voler realizzare una rete che impara la frase "the cat chased the mouse". Abbiamo una rete con un singolo strato nascosto, che imparerà per ogni parola la probabilità di quella successiva, considerando anche il passato. Il vettore w_{hh} collega i neuroni dello strato nascosto con sè stessi, al momento in cui riceveranno la parola successiva. In figura 1.9 a sinistra vediamo la rappresentazione architetturale, a destra vediamo come si comporta nella progressione dell'intera frase: a un primo istante viene sottoposta la parola "the" prevedendo in uscita "cat", al secondo istante si ha in ingresso "cat" e si prevede in uscita "chased", e così via. Osservando la

connessione w_{hh} vediamo che per ogni istante i neuroni dello strato nascosto produrranno anche un'ulteriore uscita, che costituirà lo stato iniziale all'iterazione successiva.

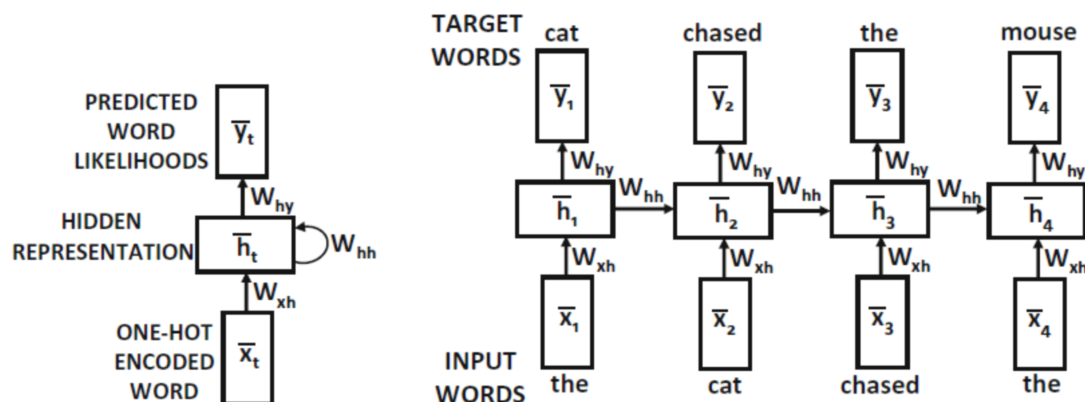


Figura 1.9: Esempio di una rete neurale ricorrente [1]. A destra è mostrata l'architettura, a sinistra il comportamento con l'andare del tempo: all'istante 1 la rete ha in input "the" e manda in input "cat", all'istante 2 ha in input "cat" e in uscita "chased", eccetera.

Questo tipo di reti è in grado di apprendere solo sequenze semplici, non è in grado di apprendere e prevedere correttamente un modello complesso che richiede di tenere conto di dipendenze di lunga distanza. Per apprendere questo tipo di modelli sono stati ideate le reti *Long Short-Term Memory* [5], caratterizzate da RNN con una funzione di risposta allo stato modificata. Questo consente di avere un controllo fine sullo stato per ottenere una memoria di lungo termine. Nella figura in seguito vediamo uno spaccato in cui si vede la differenza fra una cella RNN e una cella LSTM. Vediamo che la RNN ha una connessione diretta dallo stato precedente, mentre la cella LSTM ha diversi collegamenti, ognuno con una propria funzione attivazione e un proprio peso.

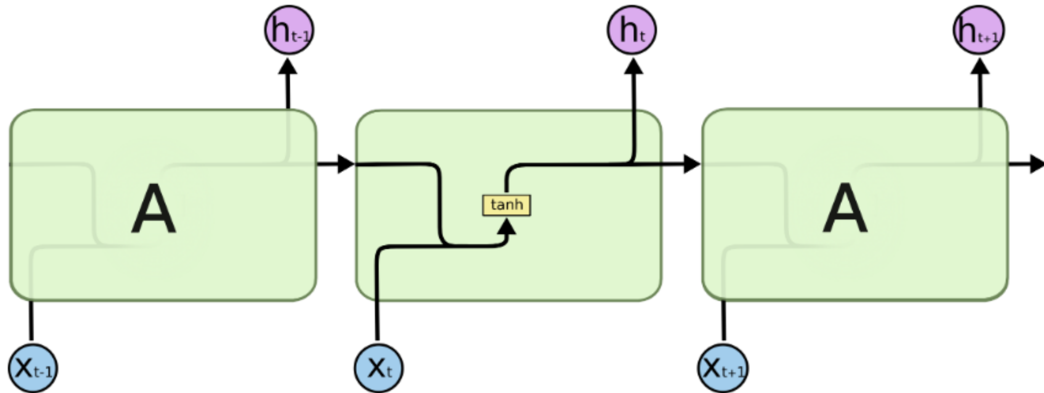


Figura 1.10: Rappresentazione con l'interno di una cella RNN. A destra abbiamo lo stato precedente, in centro lo stato corrente, a destra lo stato futuro della stessa cella [6].

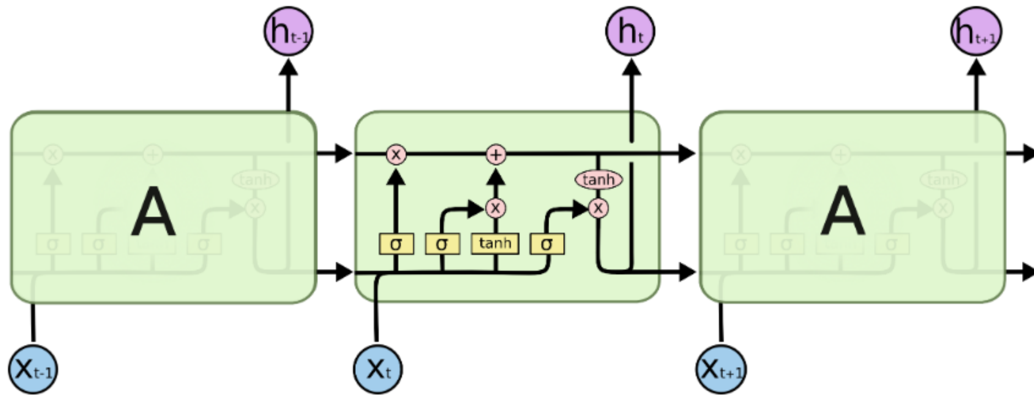


Figura 1.11: Rappresentazione con l'interno di una cella LSTM. Notare i diversi blocchi di retroazione con i pesi separati. [6].

Non è scopo di questo documento scendere nei dettagli matematici o implementativi che riguardano la realizzazione di questo tipo di reti, tuttavia si andrà a vedere l'architettura realizzata nel progetto per poter sfruttare le serie temporali derivanti da telemetrie e meteo. Questo è necessario per capire i requisiti necessari alla realizzazione dei dataset per il training di questa rete. Un concetto essenziale quando si parla di serie temporali è il *timestep*. Esso è un'unità elementare di tempo in un Recurrent Neural Network,

e caratterizza il singolo istante temporale. Nel nostro caso, l'unità elementare sarà il valore delle telemetrie e del meteo in un certo intervallo di tempo. Quello che si vuole fare è rappresentare tutti i segnali e i dati meteo sotto forma di una sequenza di valori a intervalli predefiniti, e fare in modo che la nostra rete impari a prevedere l'andamento futuro dei segnali degli impianti di interesse.

1.3.3 Il modello NMT

Il modello di Deep Neural Network realizzato per questo progetto prende ispirazione dal modello Neural Machine Translation (seq2seq) [7], nato per realizzare la traduzione automatica di frasi. Caratteristica essenziale di questo tipo di rete è la raccolta delle informazioni storiche di una sequenza per ottenere uno stato iniziale, compito realizzato da una prima porzione della rete. Questo stato iniziale viene usato per attivare la porzione di rete che esegue la previsione degli istanti successivi. Queste due porzioni vengono chiamate rispettivamente *encoder* e *decoder*. La parte encoder è costituita da un RNN con celle LSTM di cui l'uscita sarà trascurata, in modo da concentrare le informazioni storiche di più timestep in uno stato iniziale. La rete decoder riceverà lo stato iniziale e realizzerà previsioni per un numero definito di timestep. Per ogni cella successiva alla prima saranno ricevuti lo stato e la previsione ottenuti allo stadio precedente. La figura seguente rappresenta l'architettura di questa rete, si faccia attenzione ai colori delle frecce: in grigio abbiamo i vettori dei campioni al singolo istante, in rosso i vettori dello stato.

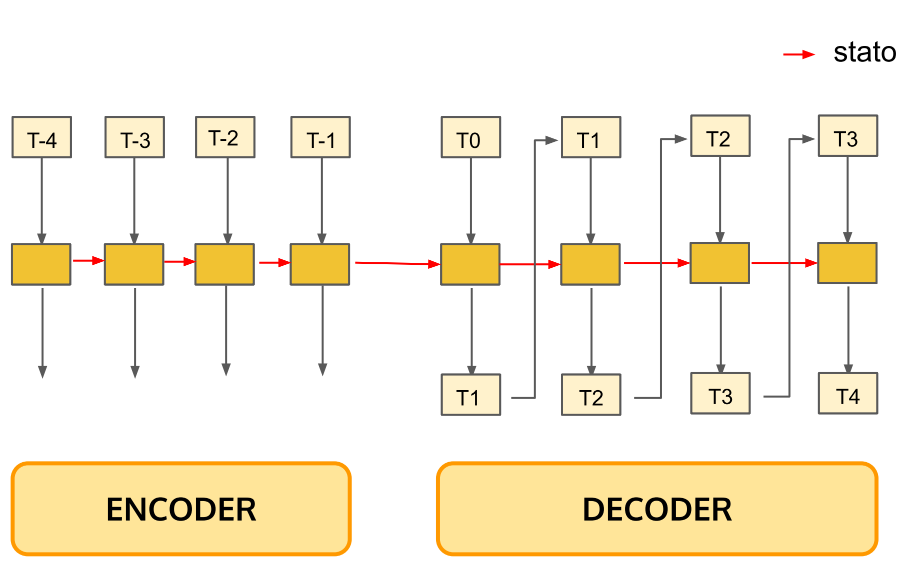


Figura 1.12: Architettura NMT. I quadrati con T_n sono i timestep, ovvero i valori a intervalli temporali fissi. I blocchetti quadrati sono le singole celle LSTM, sopra hanno il dato in ingresso e sotto il dato in uscita (la predizione). Nel decoder le predizioni sono trascurate, vengono conservati solo gli stati. Le frecce rosse sono lo stato, che viene trasferito da una cella a quella successiva. $T0$ è l'istante attuale, tutti gli istanti successivi sono predizioni. Mano a mano che il tempo scorre, il dato in ingresso scorre verso sinistra: i dati più vecchi del passato (in questa figura $T-4$ è il dato più vecchio) vengono scartati, facendo entrare da destra le informazioni più recenti.

Architettura rete previsionale

In aggiunta alla NMT, la rete progettata per il PoC di questo caso d'uso prevede di inizializzare anche la rete encoder, realizzando un ulteriore stato iniziale. L'idea è di raccogliere il maggior numero di dati possibili dal passato, considerando un istante temporale sufficientemente ampio per gli eventi di interesse, senza rendere la rete eccessivamente complessa [8]. Inoltre, le previsioni meteo delle precipitazioni vengono utilizzate come ingresso della rete decoder, considerando quindi le previsioni future di precipitazione. Esse vengono prima compresse da una rete fully connected, per ridurre la dimensionalità del dato e ridurre la complessità della rete decoder.

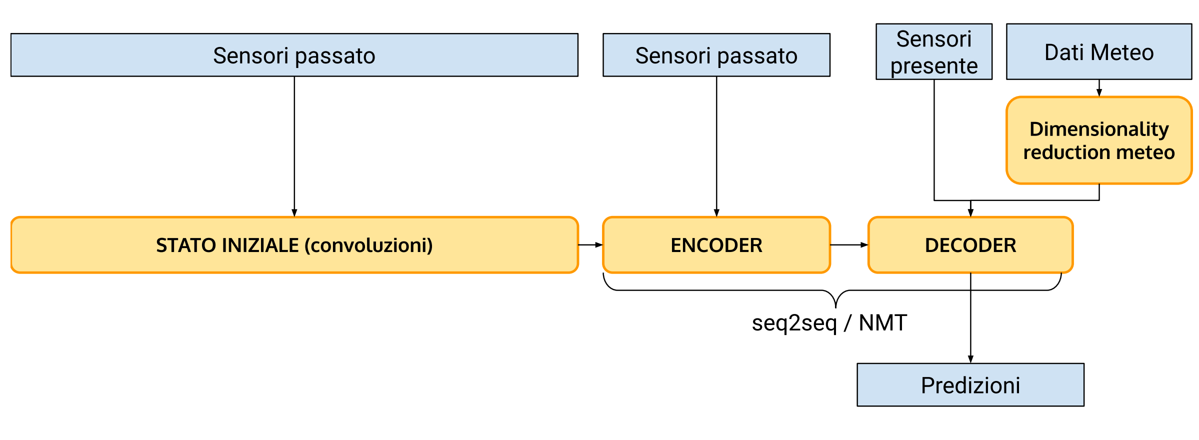


Figura 1.13: Architettura rete previsionale semplificata. A sinistra si ha la rete di generazione dello stato iniziale, che comprime le telemetrie del passato tramite strati di convoluzione. Al centro la rete encoder NMT, che riceve i dati dei sensori più recenti. A destra la rete decoder, che riceve il valore attuale dei sensori e le previsioni di precipitazione.

Lo stato iniziale dell'encoder viene ottenuto "comprimendo" i dati dai timestep precedenti attraverso un Deep Neural Network costituito da multipli strati di tipo convoluzionale. Una rete convoluzionale è caratterizzata da strati di reti neurali connessi seguendo l'operatore di convoluzione, anziché strati fully connected (dense). Queste reti sono utilizzate in ambiti dove esiste forte dipendenza locale dei dati, essendo molto più efficienti delle reti dense [1].

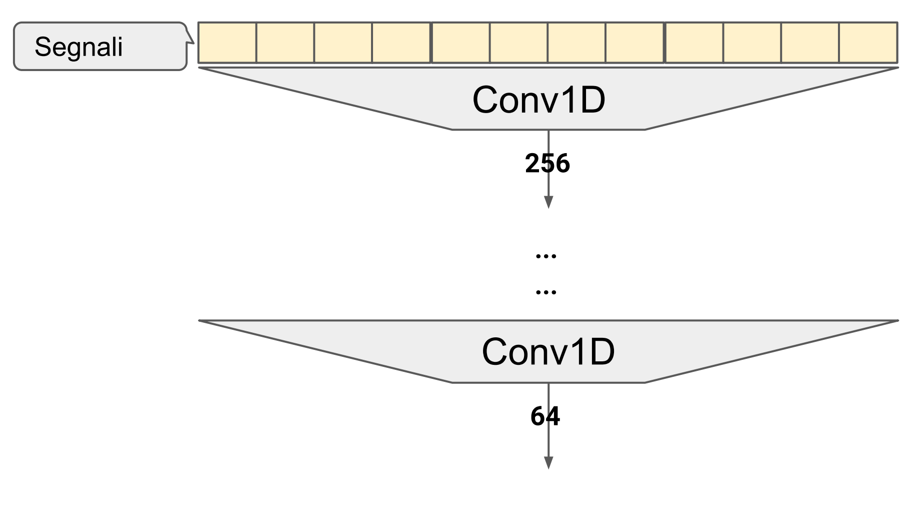


Figura 1.14: Primo stadio convoluzionale della rete previsionale. Sopra si ha un ampio intervallo temporale dei segnali, che viene compresso da un primo strato di rete convoluzionale a 256 valori. Segue con altri strati convoluzionali, fino a ottenere 64 valori. Quei valori saranno i rispettivi stati iniziali delle celle del primo timestep della rete encoder NMT.

Una volta ottenuto lo stato iniziale, che comprime le informazioni più "vecchie" dei segnali, viene alimentata la rete di encoder NMT, la quale alimenta a sua volta la rete decoder, che esegue le previsioni vere e proprie. I dati riguardanti le previsioni meteo vengono presi in considerazione solo nello stadio decoder, dal momento che non ha senso osservare i dati previsionali per eventi che sono già accaduti, avendo a disposizione informazioni in tempo reale su vento e precipitazioni. Essendo i dati delle previsioni meteo di grande cardinalità (vedi 1.2.1), viene ridotta la dimensionalità, tramite una rete multistrato fully connected. Si è scelta una rete fully connected perchè la dimensionalità dell'ingresso è ragionevole (450 valori) per cui l'utilizzo di strati convoluzionali renderebbe questa porzione di rete particolarmente ridotta. Si è ritenuto importante conservare la località dei dati, dal momento che le previsioni possono variare sia dal punto di vista spaziale che temporale.

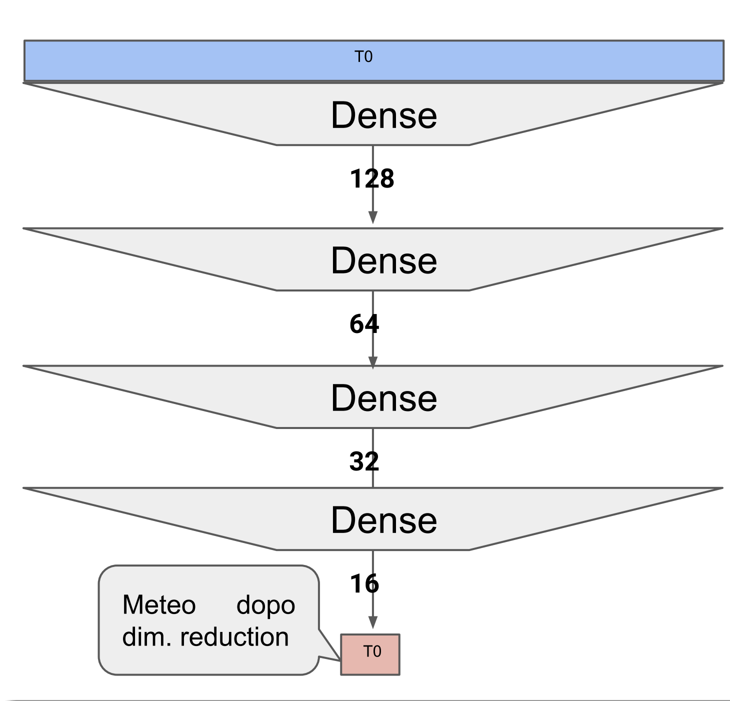


Figura 1.15: Dimensionality reduction dei dati meteo. Sopra si hanno in ingresso tutti i punti geografici con il proprio valore previsionale di precipitazione. Multipli strati fully connected riducono la dimensionalità fino ad arrivare a un vettore di 16 valori.

I fenomeni critici, di cui il cliente richiede sia previsto l'andamento e le conseguenze sugli impianti, sono i rovesci e i nubifragi. Essi sono eventi di durata breve ma intensa, soliti presentarsi ed estinguersi con rapidità giornaliera [9]. Inoltre, l'andamento dei segnali varia di una ampiezza trascurabile per intervalli inferiori ai 60 secondi. Per questi motivi, tenendo conto dei requisiti del cliente, sono state fatte le seguenti scelte per quanto riguarda il dimensionamento della rete:

- I timestep sono di 1 minuto
- Le previsioni sono di 20 minuti
- Si considera complessivamente 1 giornata ai fini delle previsioni

In seguito a primi test empirici sulle prestazioni della rete, lo stato è dimensionato a un vettore di 64 elementi, come visto in figura 1.14 e i timestep sono stati partizionati fra i tre stadi della rete nel seguente modo:

Stadio della rete	Numero timestep	Tempo
Decoder (previsione)	20	20 minuti
Encoder (dati storici recenti)	240	4 ore
Convoluzionale (dati storici lungo termine)	1180	21 ore 40 minuti

La stessa suddivisione è visibile nell'architettura in figura 1.16.

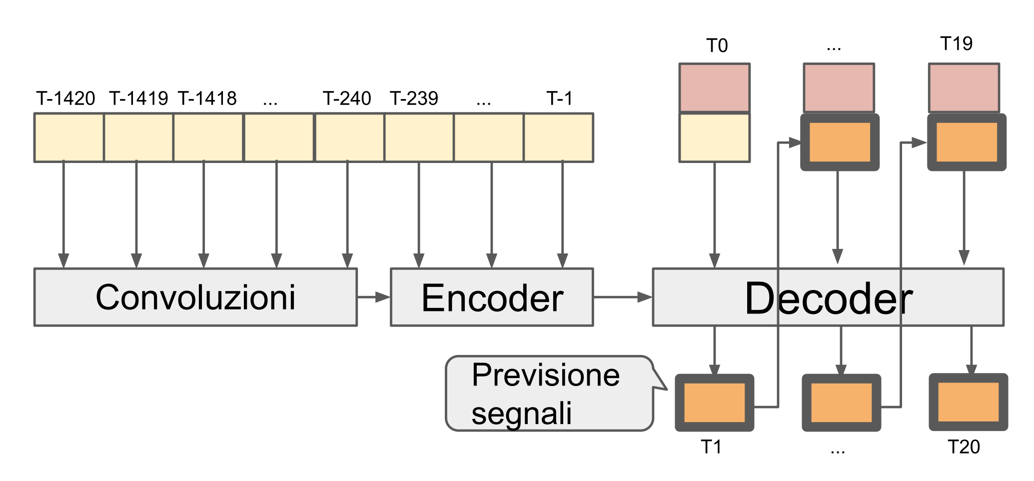


Figura 1.16: Architettura rete previsionale delle timeseries. Per ogni strato, rappresentato da un blocco intero grigio, sopra vi è l'ingresso, sotto l'uscita, a destra l'uscita dello stato finale. A sinistra si ha la rete convoluzionale del passato, che genera lo stato iniziale dell'encoder NMT. Al centro vi è l'encoder NMT, di cui si conserva solo lo stato finale. A sinistra il decoder NMT, che riceve il valore attuale delle telemetrie e del meteo, ed esegue le previsioni delle telemetrie.

Capitolo 2

Dataset e training

In questo capitolo si andrà a vedere il design architetturale della parte che si occupa di generare il dataset e il training della rete neurale. Si vedrà nel dettaglio una sua parte, i nuovi dataset generati e i risultati dei training effettuati con quei dataset. Nella seguente figura si ha uno schema rappresentativo della prima parte, che si occupa della realizzazione del dataset. Ogni parte all'interno dell'architettura è racchiusa in un blocco che rappresenta quale tecnologia è stata utilizzata per la sua realizzazione.

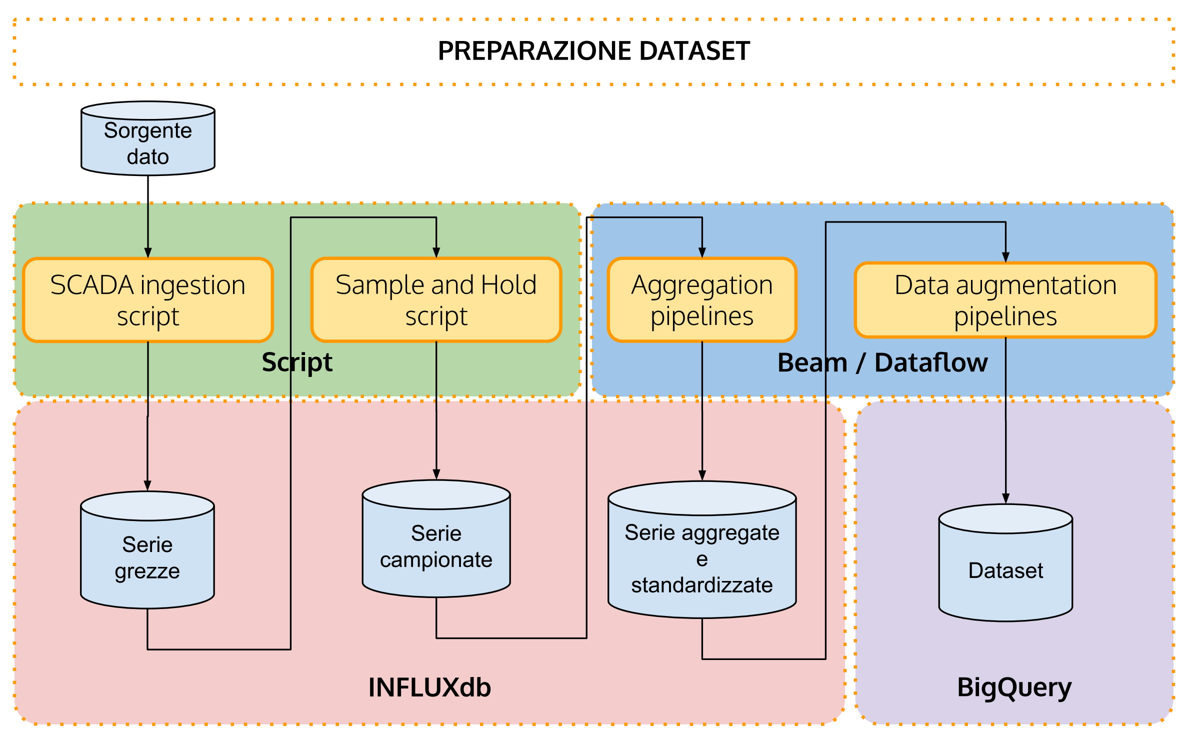


Figura 2.1: Schema architetturare generazione dataset. I cilindri rappresentano basi di dati, i blocchi gialli sono processi di trasformazione dei dati. Ogni elemento dell'architettura realizzata si trova in un piano che indica la tecnologia utilizzata per esso, tranne la sorgente dato che è del cliente e viene considerata esterna. Su Script abbiamo dei script UNIX, su InfluxDB ogni cilindro è un database diverso, su Beam/Dataflow ogni blocco è una pipeline differente, infine su BigQuery il Dataset è memorizzato su una propria tabella.

2.1 Ingestion

Per poter realizzare un progetto di Data Science utilizzando i dati provenienti dai sistemi del cliente è necessario effettuare la *ingestion* di questi dati in un *repository* accessibile in base alle necessità, conformemente al modello di Data Warehouse [10]. Questo perché in fase di creazione di un dataset è necessario accedere rapidamente a una ampia porzione dei dati, operazione che normalmente non è ottimizzata nei sistemi dedicati alla produzione e alla operatività nominale dei sistemi. I dati dello SCADA e delle precipitazioni meteo vengono memorizzati in due database differenti.

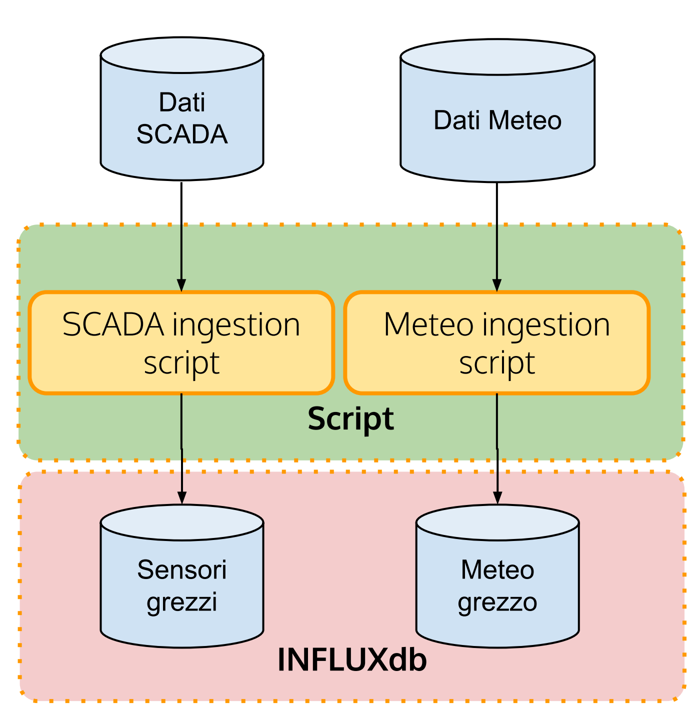


Figura 2.2: Ingestion dei dati grezzi. I dati di SCADA e di precipitazioni meteo provengono in formati diversi e hanno campi diversi, quindi vengono salvati in database differenti.

2.1.1 InfluxDB

Come si è visto, i dati di telemetrie dei sensori e del meteo sono serie temporali. Il modello di rete neurale è basato su questo tipo di dato, anche se ha requisiti specifici riguardanti il formato. È stato individuato e utilizzato un database particolare, realizzato appositamente per trattare esclusivamente le serie temporali, il quale offre già alcune trasformazioni specifiche ad esse. Le informazioni che vengono memorizzate al suo interno, infatti, rimangono legate al proprio timestamp. Il timestamp è l'istante in cui un certo dato è stato prodotto. In una serie temporale avremo una serie di campioni, ognuno con il proprio timestamp. Le astrazioni che offre InfluxDB sono:

- Database
- Measurement
- Tag

Database e Measurement potrebbero essere considerati gli analoghi, rispettivamente, di un database e una tabella di DB comune. Tuttavia, all'interno di un Measurement, si ha possibilità di inserire solo dati telemetrici, distinti tramite il Tag. Il Tag è quindi la chiave primaria all'interno di ogni Measurement. Una caratteristica importante di InfluxDB è che per ogni Tag può esistere al più una sola misura in un determinato istante. Se per qualche motivo si provano a scrivere due misure diverse dello stesso tag in uno stesso istante, rimarrà in memoria solo l'ultimo valore scritto.

2.1.2 Ingestion su InfluxDB

La fase preliminare di ingestione su InfluxDB è realizzata con script Unix che filtrano i dati dei sensori dallo SCADA e del meteo precipitazioni dell'estate 2018. Viene realizzato un Database per memorizzare i dati grezzi. Per i dati dello SCADA viene realizzato un Measurement con i dati grezzi dei sensori. Ogni sensore è identificato da un proprio "TAG", che corrisponde al Tag utilizzato come chiave primaria del Measurement. Avremo quindi un Measurement con tutte le telemetrie distinte dei sensori grezzi. I dati del meteo vengono memorizzati in un altro Measurement, utilizzando come Tag:

- Latitudine
- Longitudine
- Tipo di previsione (a 40 o 60 minuti).

Si ha quindi un Measurement contenente le serie temporali distinte per tutti i punti della regione presa in considerazione.

2.2 Campionamento e aggregazione

Come visto in precedenza, l'architettura previsionale prevede di avere un timestep di tutti i dati ogni minuto. Questo significa che per tutte le serie temporali in analisi si deve avere un campione ogni minuto, altrimenti non saremo in grado di alimentare correttamente gli input della rete neurale. Per ottenere questo risultato, è necessario eseguire un ricampionamento, dal momento che dopo l'ingestion i dati grezzi hanno campioni a frequenza arbitraria. Esso viene eseguito con una tecnica nota in telecomunicazioni ed

elettronica come Sample and Hold [11]. Questa tecnica prevede di realizzare campioni a un intervallo predefinito a partire da un dato segnale. Il valore di ciascun campione sarà determinato dall'ultimo valore letto per quel segnale.

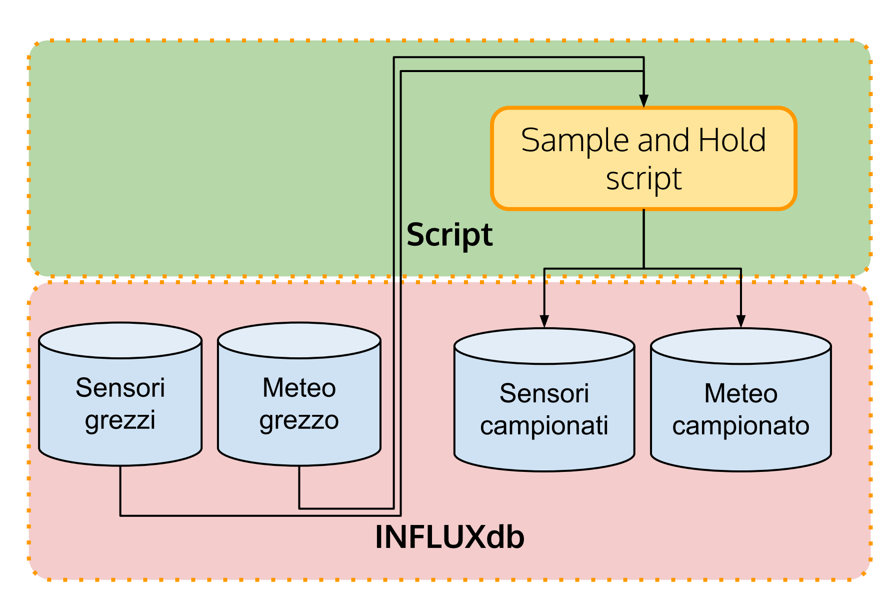


Figura 2.3: Ricampionamento dei dati grezzi. I sensori vengono campionati ogni 15 secondi, il meteo viene direttamente campionato ogni minuto.

Nella figura sotto si vede l'esempio dell'applicazione della Sample and Hold su un segnale, volendo ottenere campioni ad intervalli di 15 secondi. Sopra si ha una lettura alle 12:20:15 di valore 334. Alle 12:21:17, un'altra lettura di valore 329. Nella figura sotto abbiamo l'uscita dello stesso segnale dopo il campionamento. Ogni 15 secondi vi è un campione, che contiene l'ultimo valore aggiornato all'istante di campionamento. Dato che la lettura alle 12:20:15 avviene allo stesso istante in cui avviene il campionamento, il nuovo valore si riflette sul campione successivo delle 12:20:30. La lettura delle 12:21:17 si riflette sul successivo campione delle 12:21:30.

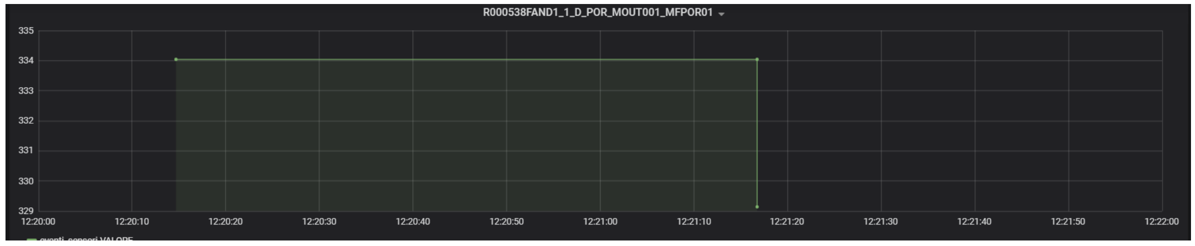


Figura 2.4: Dati grezzi del segnale. Si considera un intervallo dalle 12:20:00 alle 12:21:40. Si vedono due letture, una alle 12:20:15, una alle 12:21:28, distinguibili dai puntini verdi.

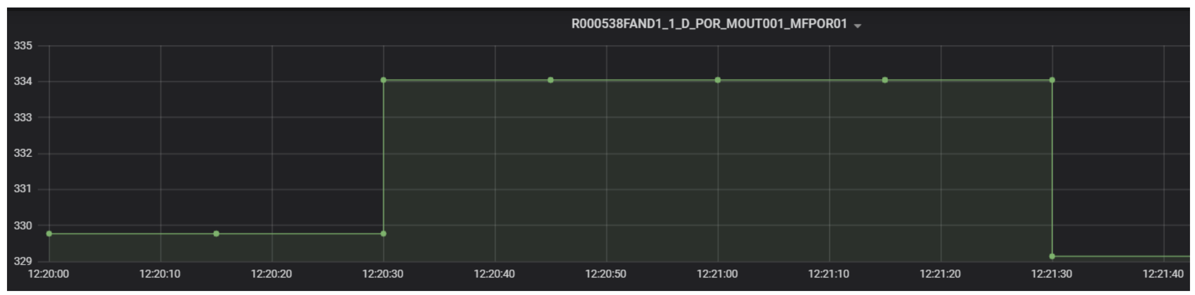


Figura 2.5: Lo stesso segnale dopo Sample and Hold. I campioni sono a intervalli fissi di 15 secondi: il primo è alle 12:20:00, il secondo alle 12:20:15, e così via. Il valore di ciascun campione è determinato dal valore della lettura più recente: alle 12:20:00 si ha il valore (non visibile nella figura sopra) della lettura precedente, che persiste fino alle 12:20:30, in cui si ha il valore letto più recentemente, e così via.

2.2.1 Campionamento

Telemetrie dei sensori

Per quanto riguarda le telemetrie dei sensori, viene eseguito un primo ricampionamento ad intervalli di 15 secondi. Il cliente ha assicurato che ciascun segnale in analisi varia con frequenza non superiore ai 30 secondi, viene di conseguenza applicato il teorema di Shannon [12] che richiede di campionare al doppio della frequenza massima del segnale di ingresso per evitare fenomeni di aliasing del segnale. In fase di studio e preparazione della rete previsionale, questa operazione viene realizzata sfruttando InfluxDB, che è in grado di realizzare autonomamente questa tecnica. Nella seguente figura si ha la query che viene realizzata in fase di ingestion considerando telemetrie dal 31-08-2018 al 04-

09-2018. Per ottenere tutti i segnali grezzi campionati secondo le specifiche, si usa la combinazione di due predicati specifici di InfluxDB:

- group by time(15s)

  Raggruppa le letture in intervalli di 15 secondi

- fill(previous)

  Gli intervalli vuoti vengono riempiti con il valore più recente

```
"select first(VALORE) AS VALORE "
"into \"" + args.influx_write_db + "\".\"autogen\".\"" + args.influx_write_measurement + "\" "
"from " + args.influx_read_measurement + " "
"where \"TAG\" = \'"+ tag +"\' "
"and \"VALORE\">=0 "
"and time >= '2018-08-31T00:00:00Z' "
"and time < '2018-09-04T00:00:00Z' "
"group by time(15s), VALORE, \"TAG\" "
"fill(previous) "
```

Figura 2.6: Query di Influx che include il predicato necessario a campionare ogni 15 secondi con metodo Sample and Hold.

Dati Meteo

Come si è visto su su 1.2.1, i dati del meteo provengono con previsioni a 20, 40 e 60 minuti, tuttavia solo le previsioni a 40 e 60 minuti vengono fornite con un anticipo ragionevole. In fase di preparazione del dataset, vengono combinati i dati previsionali a 40 e 60 minuti, conservando solamente gli intervalli validi delle previsioni a 40 minuti combinandoli con i dati previsionali a 60 minuti. Poi viene eseguito un ricampionamento con la tecnica di sample and hold già utilizzata per le telemetrie. In questo caso, il ricampionamento viene eseguito direttamente a 1 minuto, dal momento che la frequenza dei campioni del meteo è inferiore a 2 minuti.

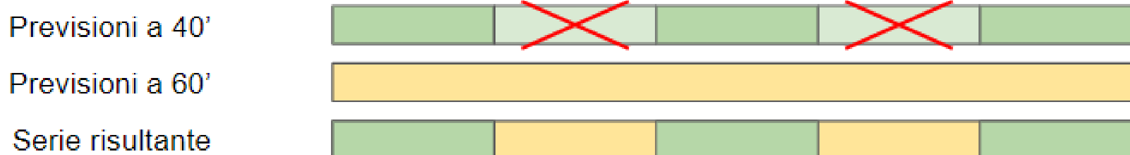


Figura 2.7: Combinazione dei dati previsionali a 40 e 60 minuti: vengono conservate solo le previsioni valide.

2.2.2 Aggregazione e standardizzazione

L'aggregazione consiste nell'elaborazione dei segnali campionati ottenuti per ottenere una rappresentazione corretta nella forma desiderata, ovvero un campione ogni minuto. Sono coinvolti i segnali delle telemetrie dei sensori, in seguito al campionamento a 15 secondi, altrimenti si perderebbe buona parte dell'informazione andando a trascurare l'andamento delle letture. Le telemetrie e i dati del meteo hanno domini di valori e distribuzioni statistiche fra loro ampiamente differenti, essi vengono riportati tutti alla stessa distribuzione statistica e allo stesso dominio, tramite un procedimento chiamato standardizzazione [13]. Questo è necessario perché le reti neurali, per funzionare correttamente, hanno bisogno che i dati in ingresso siano rapportati agli stessi domini e alle stesse distribuzioni statistiche [1].

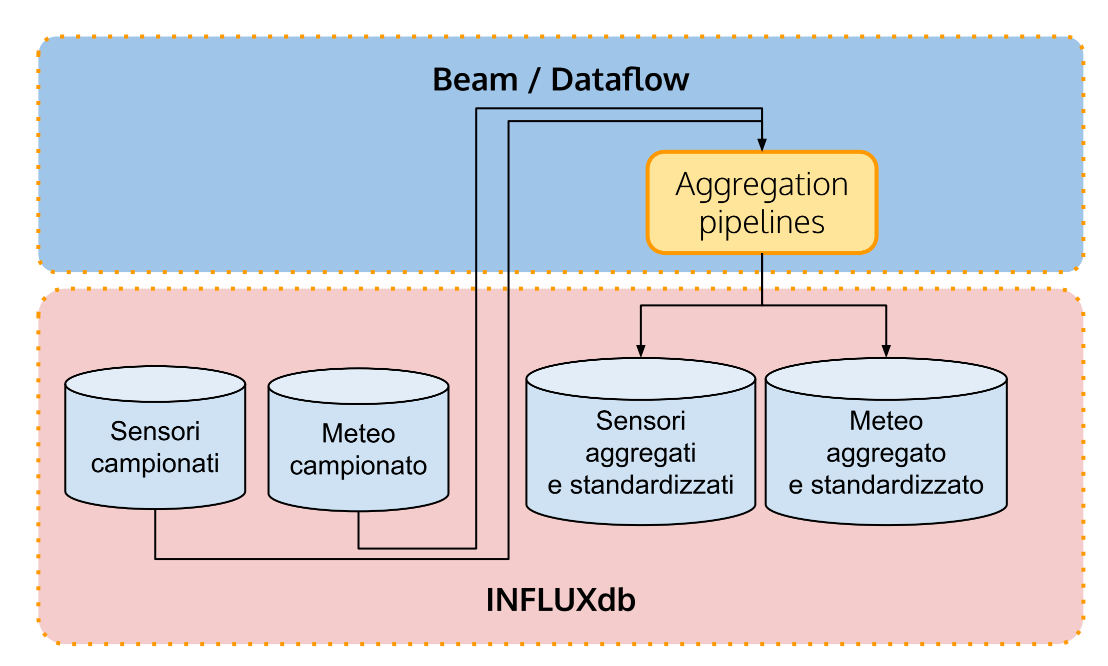
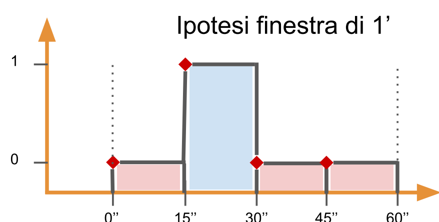


Figura 2.8: Aggregazione e standardizzazione. Per i dati dei sensori viene calcolata la media (per i segnali continui) o la moda (per i segnali discreti) per calcolare il valore ogni minuto. Sia per i sensori che per i dati meteo si effettua la standardizzazione.

Per i segnali discreti viene calcolato il valore medio nell'intervallo di ogni minuto con la regola del trapezio [14], che fornisce un'approssimazione del valore integrale in un intervallo definito. In questo modo si ottiene un valore discreto del segnale per ogni

minuto. Per quanto riguarda i segnali a valore binario, invece, viene mantenuta la moda [13], ovvero il valore più frequente nell’intervallo di un minuto.



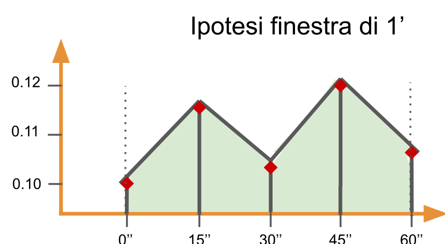


Figura 2.9: Aggregazione telemetrie sensori. Per i segnali discreti (o digitali) viene calcolata la moda nel corso del minuto. Per i segnali continui (o analogici) viene calcolata la media con la regola del trapezio.

Nel corso dello stesso processo vengono anche calcolate le statistiche per ciascuna serie temporale, ovvero la media e la varianza, per poter poi standardizzare tutti i valori in ciascuna serie temporale. Trattandosi queste ultime di operazioni con grande richiesta computazionale, che InfluxDB non è in grado di eseguire direttamente, si è scelto di utilizzare una piattaforma di processamento dei dati in grado di garantire scalabilità. Nel caso specifico è stato utilizzato Apache Beam, una astrazione a più alto livello del paradigma Map/Reduce di Hadoop, sfruttando l’esecuzione in cloud su Google Dataflow (vedi A.1.4). Si andranno a vedere alcuni dettagli di Beam nella discussione dell’architettura di realizzazione dei dati di training, realizzata utilizzando gli stessi strumenti.

2.2.3 Data augmentation

I dati delle telemetrie e del meteo vengono riuniti per formare il dataset. Come visto nel primo capitolo, l'architettura della rete neurale previsionale prevede di analizzare complessivamente l'andamento delle telemetrie in una finestra temporale di 24 ore, tenendo conto del meteo previsto nei 20 minuti di cui si vuole la previsione. Per eseguire al meglio il training della rete, una pratica diffusa è quella di aumentare il più possibile il dataset a disposizione, tramite delle tecniche note come data augmentation. Per le time series si possono ricavare finestre temporali scorrevoli, note come *time lag* [15], in modo di ottenere molte più immagini di giornate intere rispetto a quelle del dataset di partenza. L'obiettivo della rete neurale è quello di ottenere una funzione di autocorrelazione dai time lag. Nel caso in analisi, si prende prima in considerazione una finestra temporale di 24 ore (1440 minuti, corrispondenti a 1440 timestep), spostandosi poi di 1 minuto, ricavando la finestra successiva di 1440 minuti, e così via.

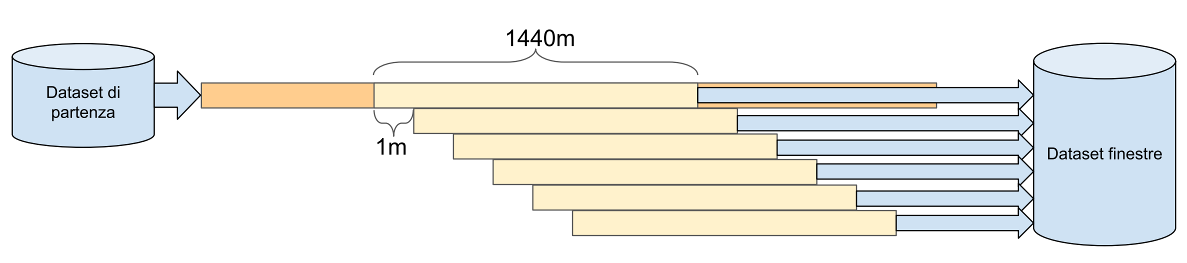


Figura 2.10: Generazione finestre e data augmentation tramite i time lag. Si parte dalla giornata normale, che inizia alle 0:00 e finisce alle 23:59. Per aumentare la dimensionalità del dataset, si considerano anche la giornata che inizia alle 0:01 e finisce alle 24:00 del giorno dopo, ma anche quella dalle 0:02 alle 24:01 del giorno dopo, e così via.

Si ottiene quindi un vasto insieme di finestre di 24 ore, dal momento che ci si sposta solo di 1 minuto per ogni finestra mobile. In questo modo, in fase di training la rete riceverà molte immagini di finestre di 24 ore.

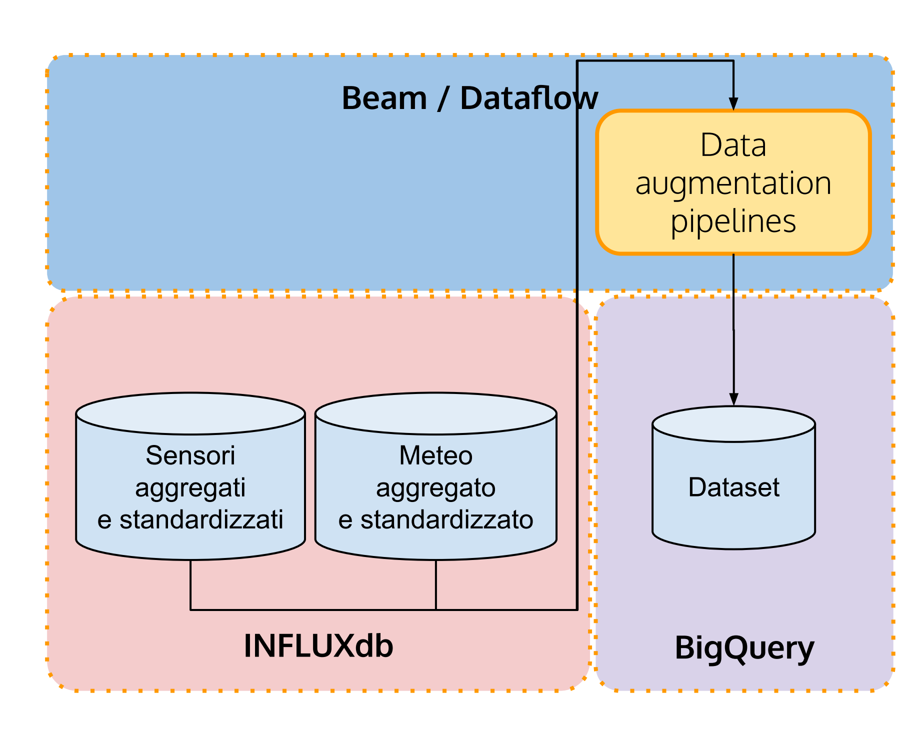


Figura 2.11: Data augmentation. Una pipeline realizzata allo scopo unisce i dati dei sensori e del meteo ed esegue il processo descritto in questa sezione e visto in figura 2.10.

2.2.4 Training della rete

Per rendere efficace il training è necessario che l'insieme delle finestre generate venga mescolato in maniera stocastica, altrimenti si rischia di incorrere in una memorizzazione progressiva delle giornate. Si tratta di un problema noto in letteratura come *Catastrophic Forgetting* [16], dove in fase di training la rete impara a svolgere un compito particolare e nel momento in cui viene sottoposta a un dataset differente perde tutto quello imparato in precedenza. Nel caso specifico, viene memorizzata una singola porzione di giornate: nel momento inizia una epoca successiva di training e quindi ricomincia il dataset, le metriche indicano un crollo repentino delle prestazioni di predizione. Questo perché il dataset è di dimensione particolarmente ampia, ma le finestre adiacenti sono molto simili. L'utilizzo di tecniche scalabili si rivela indispensabile [17], dal momento che è necessario mescolare casualmente una grande quantità di dati. Il training si esegue per un certo

numero di *epoche*, dove in ogni epoca ricomincia il dataset. Per ogni epoca è necessario ripetere il rimescolamento. È stata realizzata una architettura di preparazione dei dati di training che tiene conto di questi requisiti. Si vedrà nel dettaglio la progettazione e realizzazione di una delle parti che la compongono.

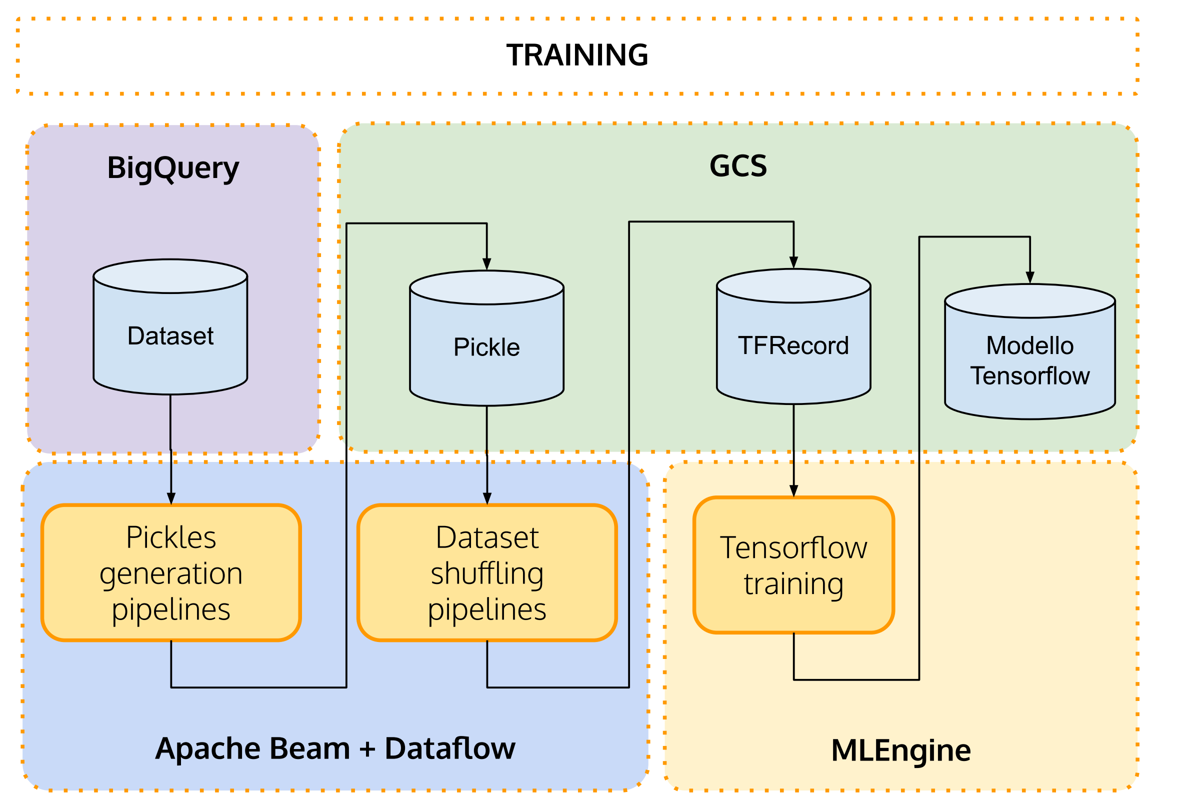


Figura 2.12: Architettura generazione dati di training e il training vero e proprio. Scopo di questa architettura è prima mescolare in maniera stocastica il dataset e poi eseguire il training, sfruttando Google Cloud MLEngine.

La prima parte consiste nella trasformazione del dataset di finestre di 24 ore memorizzate su BigQuery (vedi A.1.1: Big Query) in files del formato di serializzazione di Python denominato Pickle [18]. Questi file vengono memorizzati su Google Cloud Storage (vedi A.1.2: GCS). Viene anche realizzato un dizionario contenente gli identificativi dell'istante di inizio di ogni finestra e il corrispondente URI di collegamento al Pickle memorizzato su GCS.

Capitolo 3

Pipeline in batch e training

Si è visto che alcune fasi di processamento dei dati vengono effettuate utilizzando Beam sfruttando Dataflow, per motivi di efficienza e di scalabilità, tuttavia non è ancora stato detto nulla su cosa caratterizzi il modello per renderlo scalabile. Esso si basa su MapReduce [19], un paradigma realizzato per essere intrinsecamente parallelo, permettendo di scalare l'elaborazione di grandi quantità di dati su un sistema distribuito e limitando la capacità di computazione solamente al numero di macchine a disposizione. Ogni operazione deve poter essere definita in due funzioni, eseguite in sequenza: la funzione di Map e quella di Reduce. Una caratteristica essenziale che i dati passati a queste funzioni devono avere è quella di essere rappresentabili come coppie $< chiave, valore >$[1] . Una pipeline, in questo ambito, rappresenta una astrazione del flusso dei dati attraverso le operazioni di trasformazione [20]. Apache Beam è un framework open source per la definizione ed esecuzione delle pipeline per il processamento dei dati sia in batch che in streaming. Una pipeline in batch è caratterizzata dal fatto di avere tutti i dati disponibili al momento in cui la pipeline viene lanciata in esecuzione. Sono supportati diversi framework di esecuzione, chiamati anche *runner*. Un runner di Beam è un'interfaccia di esecuzione distribuita, che permette di mettere a disposizione anche come servizio (a pagamento) la capacità di computazione su cloud. Google Cloud Dataflow è uno dei runner supportati da Beam, che permette di sfruttare le risorse disponibili su Google Cloud.

[1] $< a, b >$ rappresenta una tupla di due elementi a, b.

3.1 PCollection

La PCollection è l'unità elementare del dato in Beam. Garantisce maggiore flessibilità sui tipi di dato che vi possono essere inseriti, dato che non limita alla coppia $< chiave, valore >$ richiesta dal paradigma MapReduce. Una pipeline elabora i dati in uno o più insiemi di PCollection, che rimarranno collegati alla stessa pipeline fino al suo termine. Le caratteristiche di una PCollection sono:

1. **Tipo**: Gli elementi di una PCollection possono essere di ogni tipo, ma all'interno della stessa collezione tutti gli elementi devono essere dello stesso tipo.

2. **Immutabilità**: Una PCollection è immutabile, ogni fase di processamento della pipeline può elaborare e produrre nuove PCollection, ma i contenuti della PCollection di origine vengono letti e non possono essere alterati.

3. **Accesso**: Gli elementi delle PCollection non possono essere letti in modo casuale, dato che Beam consuma gli elementi singoli all'interno delle PCollection. Questo significa che gli elementi all'interno di una PCollection vengono iterati in un ordine prestabilito che non può essere variato.

4. **Dimensione**: Una PCollection è un insieme di elementi di dimensione non determinabile a priori. La quantità di elementi che può essere contenuto in una PCollection non ha limiti.

3.2 Operazioni elementari

In questo paragrafo si vedranno alcune delle operazioni elementari sul framework Apache Beam, in particolare quelle utilizzate nella parte di shuffling casuale delle finestre nel dataset alla fine della realizzazione dei dati di training. Una caratteristica essenziale di tutte queste funzioni è l'atomicità. In generale, una funzione atomica $f(x)$ deve restituire sempre lo stesso risultato data una stessa x in ingresso.

3.2.1 ParDo

Una ParDo è una trasformazione generica, simile alla funzione di Map nel paradigma MapReduce. La ParDo riceve gli elementi singoli di una PCollection in ingresso ed

emette zero, uno o più elementi a una PCollection in uscita. Operazioni tipiche di una ParDo possono essere:

- Filtraggio di un dataset
- Formattazione o conversione di tipo di ogni elemento in un dataset
- Estrazione di parti del contenuto dagli elementi in un dataset
- Eseguire computazioni sui singoli elementi di un dataset

L'operazione che sarà eseguita dalla ParDo è definita in una funzione denominata *ProcessElement* che deve essere implementata dal programmatore. All'interno di tale funzione verrà concentrata la logica della trasformazione. Questo permette di separare il framework Beam dalla business logic: è possibile e raccomandato testare le funzioni di trasformazione separatamente dal resto e prima di lanciare l'intera pipeline.

3.2.2 GroupByKey

Questo tipo di trasformazione prevede di avere in ingresso PCollection di tipo

$$< chiave, valore[, valore_1, \ldots, valore_n] >$$ [2]

In uscita avremo delle PCollection del tipo

$$< chiave, Iterable < valore[, valore_1, \ldots, valore_n] >$$

dove *Iterable* <> rappresenta una collezione iterabile di elementi, per cui gli i dati in ingresso verranno raggruppati in base alla chiave. Si tratta di una operazione di aggregazione di elementi. Per avere un semplice esempio di GroupByKey, si consideri il seguente insieme di coppie $< chiave, valore >$:

```
cat , 1
dog , 5
and , 1
cat , 5
```

[2] $[, valore_1, \ldots, valore_n]$ rappresenta un insieme di elementi opzionali, in numero arbitrario.

```
dog,  2
and,  2
cat,  9
and,  6
```

Dopo l'applicazione di una GroupByKey, si hanno gli elementi aggregati:

```
cat,  [1,5,9]
dog,  [5,2]
and,  [1,2,6]
```

3.2.3 CoGroupByKey

Si tratta di un'operazione di aggregazione più complessa, dal momento che corrisponde all'unione relazionale fra due o più PCollection del tipo $< chiave, valore[, valore_1, \ldots, valore_n] >$ aventi la stessa chiave, ma valori di tipo diverso. In uscita si hanno le combinazioni degli elementi, dove quelli con la stessa chiave avranno sia gli elementi della prima collezione, sia gli elementi della seconda. Un esempio semplice si può fare considerando una rubrica di email e una rubrica telefonica:

```
emails_list = [
('amy', 'amy@example.com'),
('carl', 'carl@example.com'),
('julia', 'julia@example.com'),
('carl', 'carl@email.com'),
]
phones_list = [
('amy', '111-222-3333'),
('james', '222-333-4444'),
('amy', '333-444-5555'),
('carl', '444-555-6666'),
]

emails = p | 'CreateEmails' >> beam.Create(emails_list)
phones = p | 'CreatePhones' >> beam.Create(phones_list)
```

Dopo aver applicato la CoGroupByKey, i dati risultanti saranno la combinazione dei dati, considerando le chiavi uniche:

```
results = [
('amy', {
'emails': ['amy@example.com'],
'phones': ['111-222-3333', '333-444-5555']}),
('carl', {
'emails': ['carl@email.com', 'carl@example.com'],
'phones': ['444-555-6666']}),
('james', {
'emails': [],
'phones': ['222-333-4444']}),
('julia', {
'emails': ['julia@example.com'],
'phones': []}),
]
```

3.3 Dataset shuffling

Una versione iniziale della parte di generazione dei dati di training prevedeva di mescolare precedentemente, tramite uno script, per ogni epoca di training, il dizionario delle finestre temporali, per poi generare i dati a partire dal dizionario mescolato. Tale dizionario è memorizzato in un file csv, avente il seguente formato:

```
{ 'id_finestra_1' : 'gs://uri_pickles/id_finestra_1.pkl' }
{ 'id_finestra_2' : 'gs://uri_pickles/id_finestra_2.pkl' }
{ 'id_finestra_3' : 'gs://uri_pickles/id_finestra_3.pkl' }
```

Uno script Python genera un numero di file equivalente al numero di epoche di training richiesto, avente lo stesso formato ma con i valori mescolati casualmente utilizzando la libreria NumPy, avendo l'accortezza di impostare il seed del generatore pseudo-casuale al numero dell'epoca. Questo è necessario perché il generatore di numeri casuali di NumPy, come qualunque libreria standard, è in realtà un generatore pseudo-casuale. Di

conseguenza due generatori inizializzati con lo stesso seed forniranno in realtà la stessa sequenza di numeri [21].

```python
def ShuffleTrainingData(uri_pickles,epoch,seed):
    if seed is None:
            seed = int(epoch) + 1
    with gf.Open(uri_pickles,'r') as gcs_file:
            file_list = gcs_file.readlines()
            dictionary = {eval(row).keys()[0] : eval(row).values()[0] for row in file_list}
            keys = dictionary.keys()
            numpy.random.seed(seed)
            numpy.random.shuffle(keys)
            shuffled_dictionary = dict(zip(keys, dictionary.values()))
            with gf.Open(
                uri_pickles[:-4]+'_shuffled-'+str(epoch).zfill(3)+'.txt','w+'
                ) as gcs_file_shuffled:
                for k, v in shuffled_dictionary.items():
                    gcs_file_shuffled.write('{ "' + k + '" : "' + v + '" }\n')
                return gcs_file_shuffled
```

Figura 3.1: Funzione di shuffling casuale delle finestre in un epoca realizzata in Python. Il seed viene inizializzato al numero di epoca, dato che il generatore di numeri è pseudocasuale: usare lo stesso seed provocherebbe la generazione multipla dello stesso insieme di numeri casuali.

Una volta generato il file contenente le finestre mescolate casualmente, viene lanciata la pipeline di generazione dei TFRecord, un formato specifico di Tensorflow per memorizzare i dati di training ed evaluation [22]. La rete neurale generativa che è stata vista in 1.3.3 è implementata in Tensorflow e richiede questo formato per la lettura di questi dati.

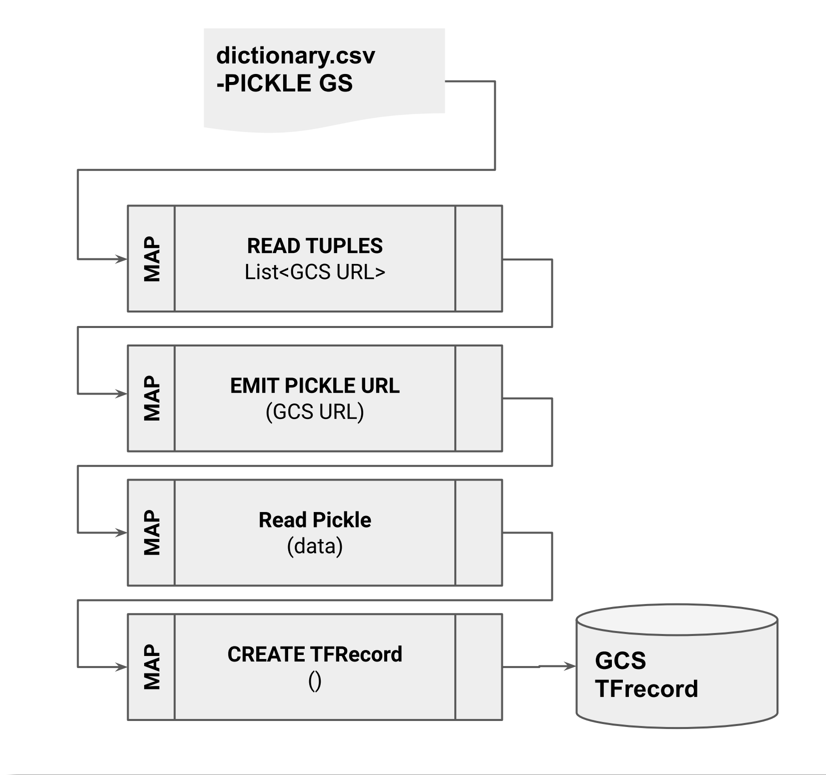


Figura 3.2: Pipeline di generazione TFRecord di un'epoca. Sopra si ha il file di origine, che contiene un dizionario di finestre di 24 ore generate come indicato su 2.2.3. I blocchi rettangolari sono stadi della pipeline. Il primo legge le singole tuple del dizionario, che è già stato mescolato casualmente. Il secondo emette per ogni tupla il suo URL, il terzo legge il file indicato nell'URL. Il quarto stadio, infine, scrive il contenuto dei file su un TFRecord, rappresentato come un cilindro.

Per ogni epoca, vengono caricati tutti i file contenenti i Pickle, nell'ordine rimescolato, per poi scriverli su dei TFRecord. Osservando il flusso di esecuzione su Dataflow, si può vedere che la parte più impegnativa è data dal caricamento dei file Pickle:

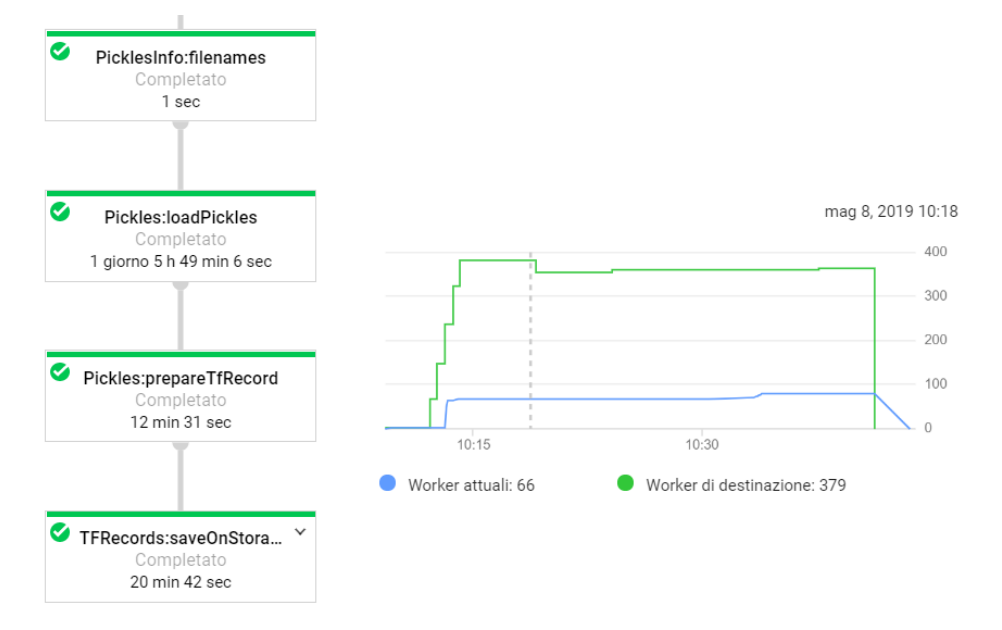


Figura 3.3: Schermata di Dataflow. A sinistra, per ogni stadio della pipeline, è indicato il carico di lavoro richiesto in termini di ore per una CPU singola. A destra, il numero di macchine impiegato per il calcolo distribuito nel corso del tempo di vita della pipeline, distinguendo fra le macchine richieste e quelle effettivamente disponibili. La pipeline è durata mezz'ora, utilizzando contemporaneamente al massimo 70 CPU circa, anche se avrebbe potuto utilizzarne 380. Questo è accaduto perchè la stessa pipeline è stata lanciata insieme ad altre 9, superando il limite complessivo di processori disponibili.

Si può notare che il caricamento dei Pickle richiede circa 30 ore di elaborazione, che si riflettono su una scalabilità effettiva a circa 400 macchine. Questa soluzione è poco efficente, dato che richiede di lanciare una pipeline diversa per ogni epoca, operazione che deve essere ripetuta anche centinaia di volte per effettuare training avanzati. I dati dei pickle, inoltre, non cambiano, cambia solo l'ordine in cui vanno scritti sui TFRecord, operazione di gran lunga meno impegnativa osservando il carico di lavoro richiesto. Si sono pensate quindi delle architetture per rendere questa operazione ancora più scalabile.

3.3.1 Soluzione parallelizzata iniziale

Inizialmente viene letta la lista degli URI dei Pickle da leggere. Dopodichè, vengono generate tante pipeline quante sono le epoche richieste. Per ogni epoca, vi è una ParDo

che esegue la funzione di mescolamento. Tale funzione, nella versione non parallelizzata, veniva lanciata a priori su uno script. Dopodichè un'altra ParDo riceve la lista mescolata dei pickle ed emette, uno per volta, gli URL da cui leggerli. Nella prossima figura si ha lo schema architetturale di questa soluzione:

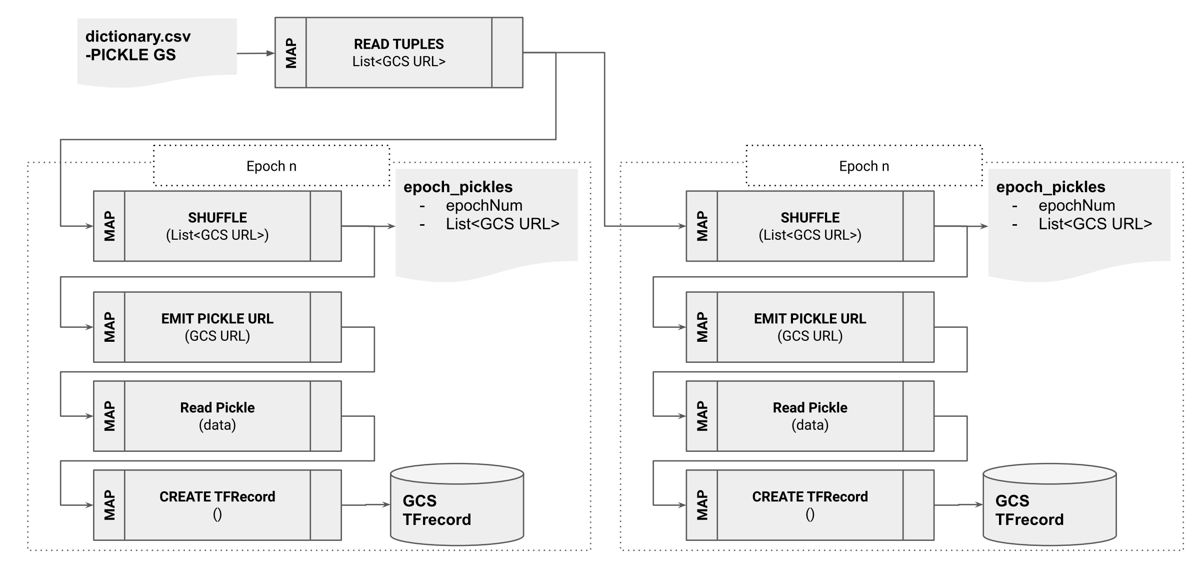


Figura 3.4: Pipeline di generazione TFRecord di un'epoca. Si ha sempre il file di origine delle tuple con le finestre e l'URL del file che la contiene, però non sarà mescolato. Lo stadio successivo legge quelle tuple ed emette gli URL. A quel punto la pipeline si divide per il numero di epoche richiesto, per semplicità ne sono raffigurate due. Per ogni epoca vengono letti tutti gli URL, rimescolati, poi vengono emessi per essere letti e scritti su TFRecord.

Tale soluzione, purtroppo, non scala adeguatamente, come si può vedere nella figura seguente:

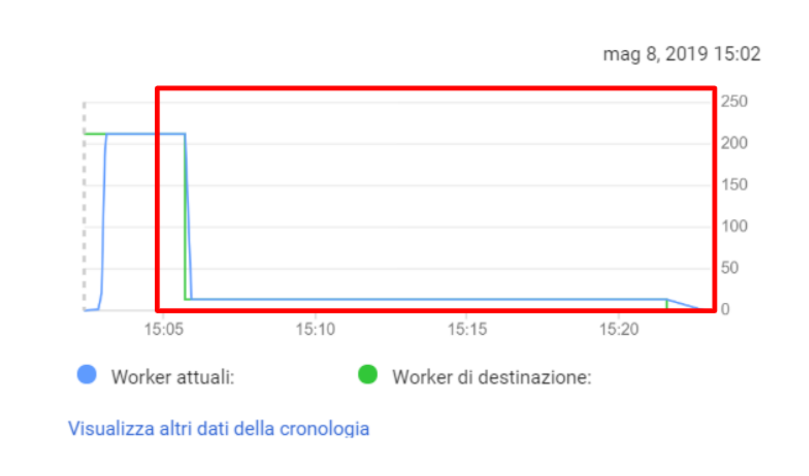


Figura 3.5: Schermata di Dataflow che rappresenta il numero di macchine usate per la pipeline. Il riquadro rosso in aggiunta evidenzia il calo repentino a 10 macchine richieste e utilizzate.

Il carico di lavoro viene automaticamente limitato a 10 macchine, corrispondenti al numero di epoche, ovvero il numero di rami della pipeline impostati nell'esecuzione in figura 3.5. Di conseguenza, l'operazione di lettura dei Pickle avviene in maniera sequenziale rispetto a ciascuna epoca. Questo perché gli URL da cui leggere i pickle vengono emessi uno per volta dalla ParDo precedente, all'interno di un ciclo, operazione che non viene ottimizzata per l'esecuzione in scala. Nella figura seguente si vede la sezione dello schema architetturale di una epoca singola, con evidenziati i punti critici:

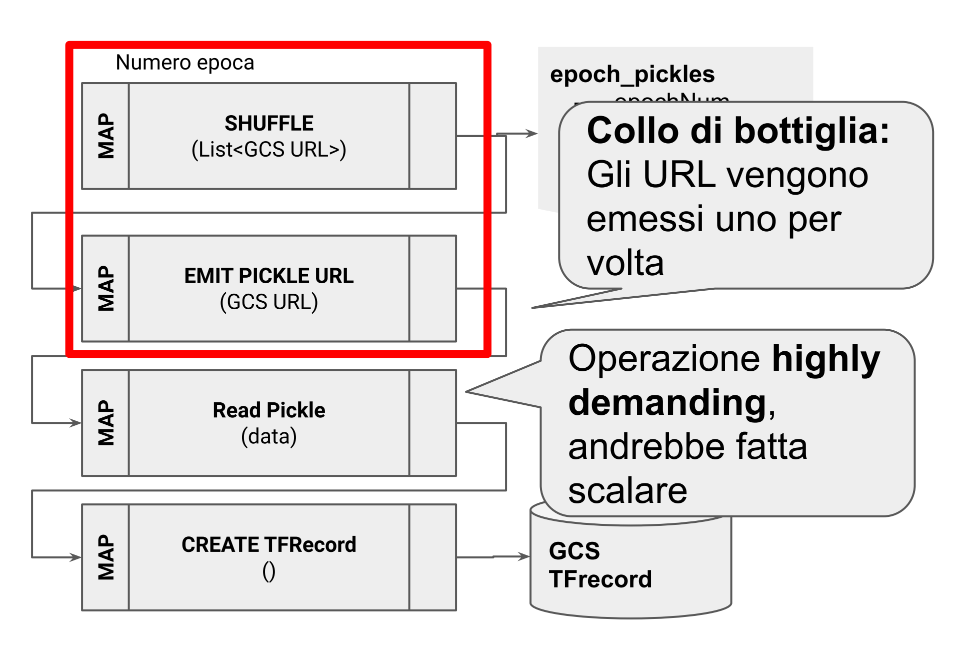


Figura 3.6: Sezione della figura 3.9 della parte con la singola epoca. Il problema è dato dal fatto che prima vengono mescolati tutti gli URL, poi lo stadio successivo li legge per emetterli uno per volta. Questa operazione non è scalabile, dato che esegue iterativamente. Lo stadio successivo, che legge i file, è molto onerosa e per essere eseguita in tempi ragionevoli richiede di essere eseguita in modo distribuito.

3.3.2 Soluzione parallelizzata scalabile

È stata quindi pensata una versione differente, più complessa, per separare il compito della lettura dei Pickle da quello del loro mescolamento.

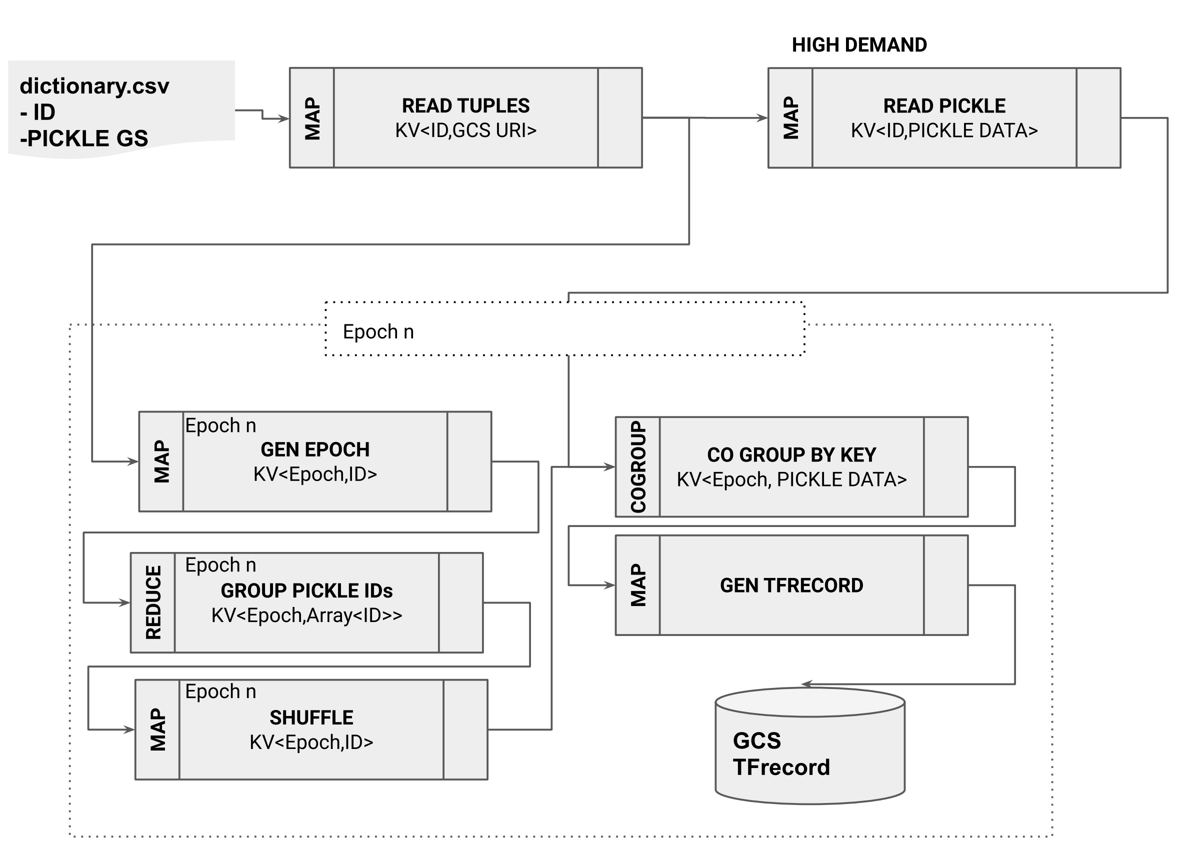


Figura 3.7: Pipeline di generazione TFRecord, versione scalabile. Come quella vista in precedenza, vengono create più derivazioni a partire dalla pipeline iniziale. Essendo più complessa, le figure a seguire analizzeranno passo per passo la sua composizione.

Con questa architettura si hanno, nuovamente, un numero di pipeline che partono dalla prima in base al numero di epoche richieste. Questa volta, tuttavia, la lettura dei Pickle viene eseguita "a monte", a ciascuna singola pipeline spetta il compito di generare un insieme differentemente mescolato di identificatori della finestra. Questi identificatori, che nella versione precedente venivano scartati, verranno usati invece come chiave in una CoGroupByKey per riunire i dati letti dai Pickle con la lista mescolata dell'epoca. Si può vedere nel concreto la parte centrale dell'implementazione, ovvero la pipeline, seguendo il percorso dello schema ideato. Ogni pipeline inizia partendo da un oggetto Pipeline di Beam, poi abbiamo la lettura del CSV con una funzione predefinita di Beam, seguita da una ParDo che estrae le tuple $< id_finestra_n, URI_n >$. Queste tuple vengono inserite in una PCollection denominata $pickleNames$. Questa PCollection viene fornita in input

a una ParDo che legge i Pickle relativi ai corrispondenti URI, e manda in uscita tuple $< id_finestra_n, PICKLE_DATA >$ in una PCollection denominata $picklesTrain$. È importante notare che le PCollection in uscita da Read Pickle conterranno interi file, tuttavia non è un problema grazie alle caratteristiche delle PCollection.

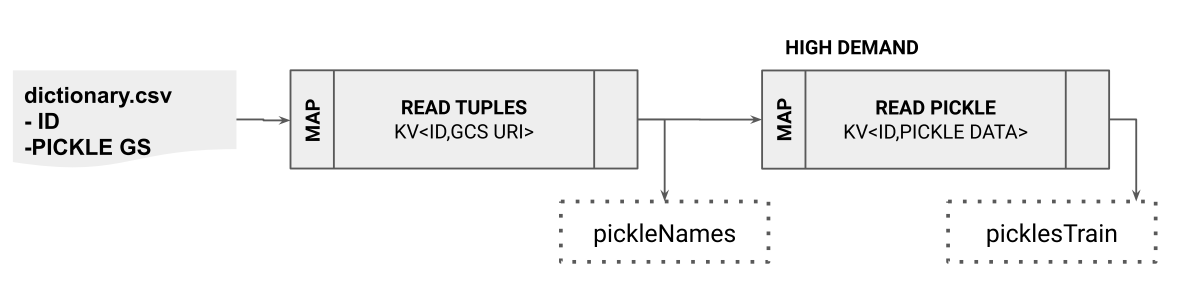


Figura 3.8: Lettura delle tuple e dei Pickle: viene letto il dizionario delle finestre di 24 ore, di cui viene conservato sia l'identificativo sia l'URL. Un primo stadio emette queste singole tuple senza modificarle, quello successivo per ciascuna tupla emette l'identificativo e il contenuto del file nel suo URL. In questo modo la lettura viene eseguita preventivamente, conservando una chiave per ogni file che è stato letto.

Qui sotto si vede una porzione di codice Python riguardante solo la dichiarazione della Pipeline e delle funzioni da applicare in ogni passaggio. È importante osservare che questo permette di avere una netta distinzione fra la *business logic* delle singole operazioni e il flusso dei dati nelle pipelines.

```
# pipeline
p = beam.Pipeline(argv=argv)

pickleNames = (p | 'ReadFromText'  >> beam.io.ReadFromText(opt.uri_pickles_train)
                 | 'ReadTuples' >> beam.ParDo(ReadTuples())
              )

picklesTrain = (pickleNames
                | 'ReadPickle' >> beam.ParDo(ReadPickle()) )
```

Figura 3.9: Pipeline in Python. La prima parte è predefinita e definisce l'oggetto Pipeline. La riga che inizia per pickleNames è la parte di pipeline che esegue la lettura dei nomi dal file di testo e ne estrae le tuple con la ParDo definita nella funzione ReadTuples, omessa in questa figura. La riga con picklesTrain prende il risultato della precedente, per leggere il contenuto dei file negli URL, con la ParDo definita su ReadPickle, anch'essa omessa.

A questo punto, per ogni epoca, viene generata una pipeline. Per quanto riguarda pickleNames, essi vengono rimescolati per ogni epoca in tre passaggi:

1. Si genera una coppia $< epoca, id_finestra_n >$ per ogni $id_finestra_n$
2. Si raggruppano con una GroupByKey tutti gli $id_finestra$
3. Gli $id_finestra$ vengono rimescolati e vengono emesse tuple del tipo $< epoca, id_finestra_n >$, in ordine sparso rispetto a quello di partenza.

Queste tuple vengono assegnate alla PCollection $shuffledEpoch$.

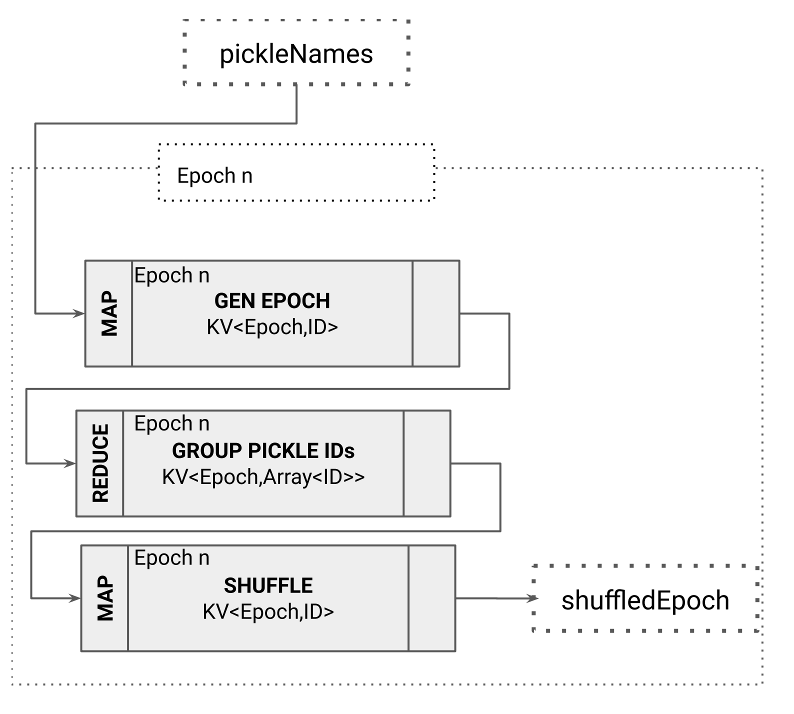


Figura 3.10: Shuffling della singola epoca. Il primo stadio riceve gli identificativi delle finestre e li associa al numero di epoca del ramo. Ogni ramo della pipeline avrà un identificativo di epoca proprio. Gli identificativi della finestra vengono raggruppati in un'unica collezione, poi vengono mescolati casualmente usando come seed l'epoca corrente. La collezione di identificativi viene iterata, in modo di avere in uscita delle tuple composte dal numero di epoca e un identificativo. Gli identificativi in uscita su shuffledEpoch sono mescolati casualmente rispetto a quelli in ingresso, in ordine, su pickleNames.

A questo punto si ha a disposizione la PCollection $picklesTrain$ con il contenuto dei

Pickle associato alla loro chiave e un'altra PCollection $shuffledEpoch$ con il progressivo dell'epoca e la chiave dei Pickle, ma in ordine diverso. Avremo tutti gli ID dei pickle ripetuti per il numero delle epoche. Con una GroupByKey si vanno, in un solo colpo, a raggruppare le epoche e a riunire il contenuto dei Pickle:

```
shuffledEpochData = ({ 'key': shuffledEpoch , 'data': picklesTrain }
                     | 'CoGroupByKey'+epoch_str >> beam.CoGroupByKey()
)
```

Figura 3.11: Parte di codice Python della pipeline, particolare con la CoGroupByKey. Vengono dichiarate come dizionario le due PCollection di cui si vuole effettuare unione, poi si chiama la CoGroupByKey.

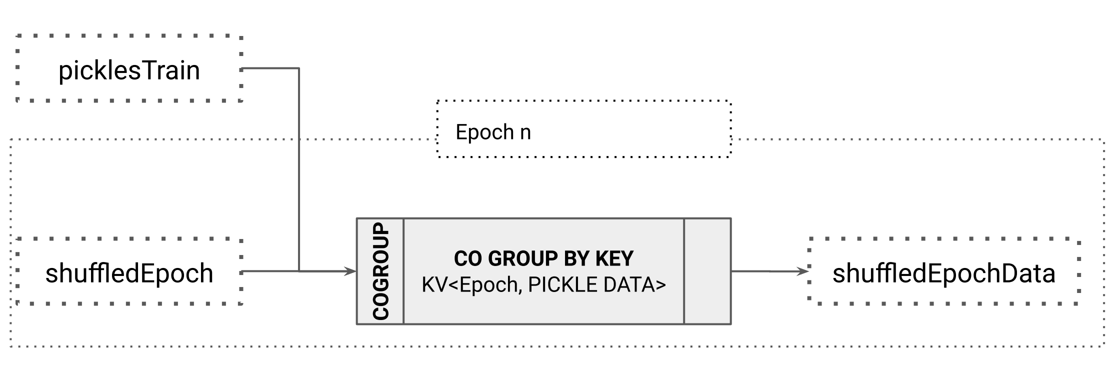


Figura 3.12: CoGroupByKey fra le chiavi mescolate e i dati dei Pickle caricati. A sinistra vengono prese da picklesTrain le tuple con identificativi e file già letti dai Pickle, da shuffledEpoch le tuple formate da numero epoca e identificativo finestra. Si otterrà una collezione con i dati letti dai file mescolati in ordine diverso.

Infine, avverrà la scrittura dei dati su TFRecord, sempre all'interno della pipeline separatamente per ogni epoca. Questa soluzione garantisce un'alta scalabilità ed efficienza, dal momento che l'operazione più dispendiosa (la lettura dei Pickle) viene eseguita una volta sola e i suoi risultati trasmessi a tutte le pipeline che rimescoleranno e scriveranno i TFRecord di ciascuna epoca. Nella prossima figura si ha un ritaglio della schermata di Dataflow sulla pipeline lanciata per la generazione di 50 epoche. Si noti la ReadPickle, che è collegata alle CoGroupByKey di tutte le 50 epoche. Subito sotto si vede l'altro ingresso della CoGroupByKey, ovvero shuffledEpoch.

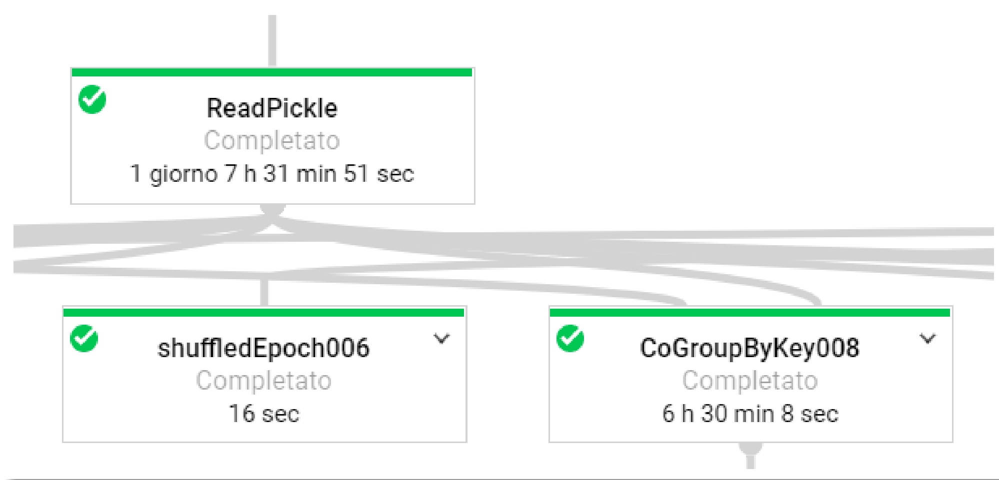


Figura 3.13: Sezione della rappresentazione a blocchi della pipeline su Dataflow. Per ogni blocco, è indicato il tempo CPU complessivo utilizzato. Il compito della lettura dei Pickle, il più oneroso, è stato eseguito una volta soltanto, e viene sfruttato per tutte le epoche della pipeline.

Come si può notare, questa pipeline ha scalato con successo al massimo numero di worker possibili. La generazione di 50 epoche, con gli stessi dati e utilizzando la prima versione della pipeline che genera i TFRecord singolarmente avrebbe richiesto 200 minuti complessivamente. Con questa pipeline si impiega $\frac{1}{3}$ del tempo e vengono consumate $\frac{1}{4}$ delle risorse.

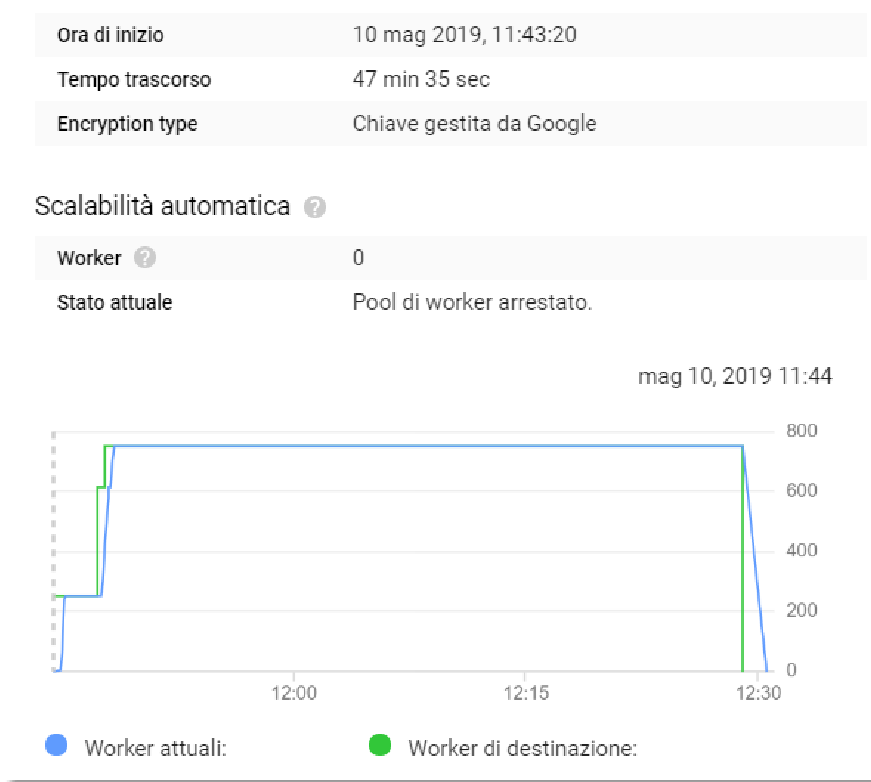


Figura 3.14: Porzione di schermata di Dataflow con l'allocazione del calcolo distribuito. Sono stati sfruttati tutti i processori a disposizione della piattaforma, per quasi tutto il tempo di esecuzione.

Non tutto è perfetto: un problema da non trascurare è che la CoGroupByKey perde l'ordine di entrambe le PCollection in arrivo. Di conseguenza l'ordine delle finestre nei TFRecord risultanti non sarà quello previsto in fase di shuffling.

3.4 Training della rete

Per il training della rete neurale, realizzato con tecnologie Cloud utilizzando Google ML Engine (vedi Appendice A.1.5: IA Hub), sono stati provati diversi parametri riguardanti il fine tuning della rete, con diverse configurazioni, utilizzando due dataset in particolare. Si andranno a vedere i risultati più rilevanti delle prove con i due dataset realizzati.

3.4.1 Modalità di training della rete

La metrica utilizzata per valutare la bontà dei risultati è il Mean Absolute Error (MAE) [3]. Il Mean Absolute Error è un'alternativa al Mean Squared Error, più robusta rispetto agli outlier, ovvero quelle istanze per cui l'errore di predizione è superiore rispetto a tutte le altre. Il MAE è definito come:

$$\frac{|p_1 - a_1| + ... + |p_n - a_n|}{n}$$

dove p_i è il valore predetto e a_i il valore atteso. Il suo valore è da 0 a $+\infty$, dipendentemente dai dati. Per il training sono state utilizzate due modalità principali, con diverse varianti della funzione obiettivo. Queste modalità sono state chiamate "direct" e "feedback". Nella modalità "direct" abbiamo che per ogni timestep di predizione viene dato in input il valore del training set. Ogni valore predetto sarà confrontato con il valore obiettivo del training set. Nella prossima figura si vede una rappresentazione architetturale della rete in fase di training direct. Si faccia riferimento all'architettura previsionale presentata su 1.3.3: Architettura rete previsionale.

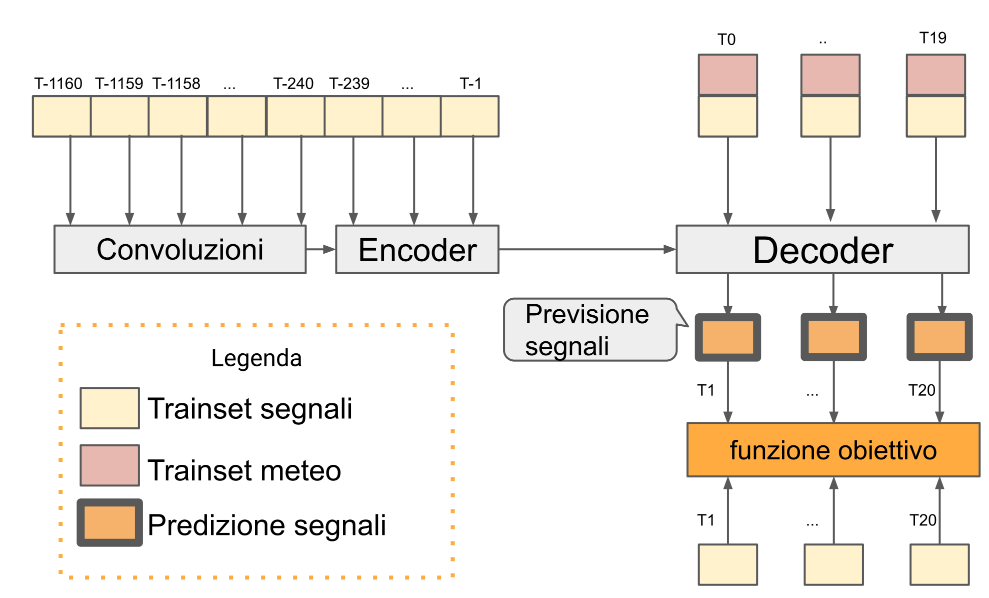


Figura 3.15: Training in modalità "direct". Sopra si ha l'ingresso del training set, anche per i timestep in cui la rete dovrebbe fare le previsioni. Sotto viene eseguito il confronto della funzione obiettivo per ogni timestep fra la previsione eseguita e quella ottenuta per la rete.

In modalità feedback invece i valori predetti vengono utilizzati allo stesso modo in cui andranno utilizzati dopo il training, ovvero come valore di ingresso al timestep successivo. La funzione obiettivo confronterà sempre la previsione di ogni timestep con i valori attesi nel training set.

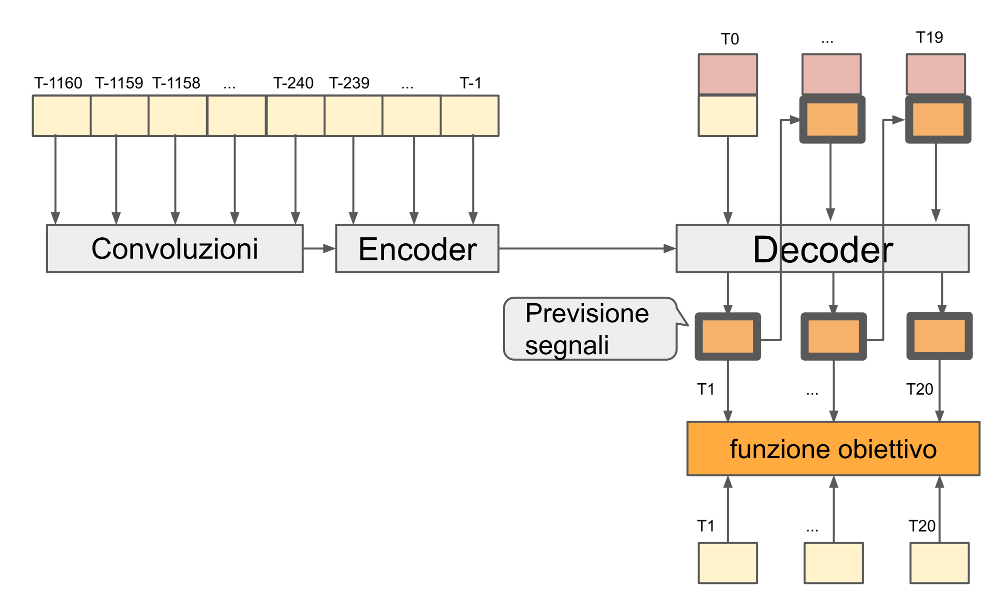


Figura 3.16: Training in modalità "feedback". Vengono forniti i dati del training set alla rete solo fino al timestep "attuale", che sarebbe il 1421° minuto della giornata nella finestra di 24 ore presa in considerazione, ricordando che il dataset è formato da finestre di 24 ore composte di 1440 minuti (timestep). Le previsioni della rete sono usate per ottenere i valori in input successivi, allo stesso modo in cui farebbe le previsioni dopo il training. I risultati delle previsioni vengono confrontati con quelli attesi attraverso la funzione obiettivo.

3.4.2 Risultati con dataset estate 2018

Questo dataset è costituito da un training set comprendente tutti i dati da Giugno ad Agosto 2018 e un evaluation set costituito dai dati di Settembre 2018. È emerso in generale che la rete è assimilabile a un *persistence model* [23], ovvero tende a ripetere semplicemente i dati che gli vengono forniti in ingresso: per ogni segnale, la previsione è che non varierà significativamente, indipendentemente dalle previsioni e dalle precipitazioni in corso. In seguito si ha un caso in cui il fenomeno viene evidenziato. Nella figura 3.17 si hanno i dati previsionali, dove sono previste delle precipitazioni intense fino alle

19 circa. Nella figura 3.18 si hanno due fra il sottoinsieme di segnali cui è richiesta la previsione. Il tratteggio arancione e blu è il livello nel training set, le linee continue gialle e blu invece sono l'uscita della rete. Per i primi 20 minuti la previsione è di un rialzo trascurabile dei livelli, così per i successivi 20 minuti, corrispondenti ai 20 timestep di previsione. Poi si passa alla predizione successiva, quando già i livelli sono cresciuti: la rete legge tali livelli e li mantiene allo stesso modo per tutta la durata della previsione. La stessa situazione si ripresenta per tutto il periodo relativo alla precipitazione, sbagliando di fatto le previsioni nel corso degli eventi di maggior interesse.

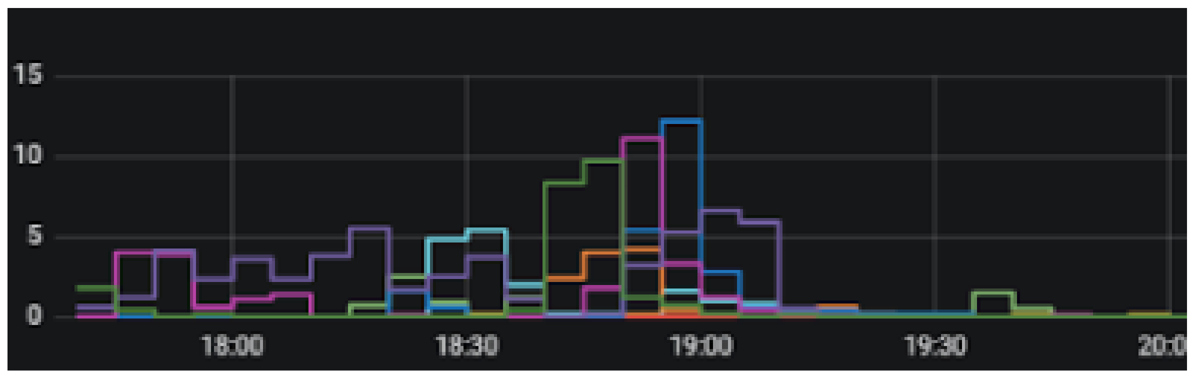


Figura 3.17: In questo grafico si vedono i livelli previsti di precipitazioni dalle 17:30 alle 20:00, in cui si ha un evento di interesse. Si verificano precipitazioni dalle 17:30, che salgono progressivamente fino a 20 millimetri alle ore 19. Dopo le 19:15 i livelli di precipitaizone scendono significativamente.

Figura 3.18: In questo grafico si vedono due fra i segnali di cui è richiesta la previsione, nello stesso intervallo della figura sopra. In rosso è evidenziata la regione critica, data dalla mancata previsione del rialzo imminente dei livelli di due segnali. Le precipitazioni della regione in analisi sono iniziate alle 17:30, con un picco intensivo alle 19. Questo picco ha provocato un innalzamento dei livelli nel giro di 30 minuti. La rete avrebbe dovuto prevedere questo innalzamento, che tuttavia non è stato previsto. Si vede un rialzo improvviso della previsione dopo 20 minuti, comportamento tuttavia non desiderato dato che si tratta della previsione successiva: la rete tende a mantenere stabili i livelli dei segnali.

3.4.3 Risultati con dataset pioggia 2018

In seguito ai risultati dei training effettuati con il dataset costituito da finestre temporali estratte da tutta l'estate, sono state effettuate delle considerazioni. In particolare, si è ritenuta l'eventualità che la rete avesse recepito un persistence model per il fatto che per buona parte degli esempi non vi è pioggia, quindi effettivamente non vi sono criticità e variazioni improvvise dei livelli dei sensori. Si è pensato quindi di individuare e ricavare delle finestre temporali dal dataset, considerando solo il sottoinsieme di giornate in cui vi sono stati eventi di pioggia. Il dataset è costituito da 41 giornate, prese considerando 8 eventi di pioggia e includendo le giornate successiva e precedente a quelle di pioggia.

Training direct

In prima analisi si verificano i risultati della rete allenata in modalità direct. In seguito si hanno delle rappresentazioni su Tensorboard (si veda A.2) del valore complessivo del MAE nel corso del training, in blu abbiamo l'andamento del training, in arancione l'evaluation:

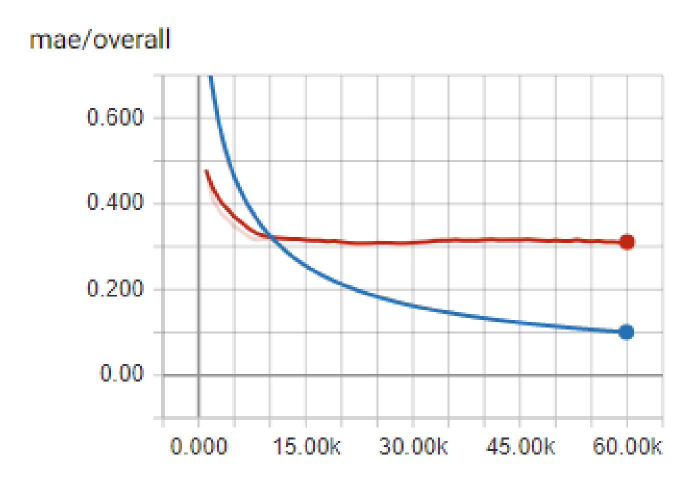


Figura 3.19: Risultati del MAE su Tensorboard per il training "direct". In blu training, in arancione evaluation. Il training continua a scendere, quindi la rete continua a imparare. Il valore medio si attesta su un errore dell'1%.

Come detto prima, il valore del MAE dipende dall'intervallo dei dati, quindi andrebbe analizzato il suo valore per ogni singolo segnale al fine di determinare quanto sia esattamente l'errore medio in valori scalari rispetto alle metriche delle singole telemetrie,

si attesta in generale a un valore medio di errore dell'1%. Risulta utile per valutare in generale l'andamento del training, per determinare se l'errore complessivamente sta diminuendo come previsto. Un'altra metrica presa in considerazione è la tabella di confusione, confrontandone i valori nel corso dell'evoluzione del training. La tabella di confusione viene costruita considerando una classificazione su attributi nominali (o categorici), come "si" e "no", e confrontando il valore previsto con quello predetto. Una buona tabella di confusione ha la diagonale principale con valori vicini al 100%, e gli altri molto bassi, a indicare che le classi previste sono quelle che vengono effettivamente predette.

Table 5.3 **Different outcomes of a two-class prediction.**

		Predicted class	
		yes	no
Actual class	yes	true positive	false negative
	no	false positive	true negative

Figura 3.20: Tabella di confusione fra due categorie [3].

Dal momento che si stanno considerando valori discreti, per generare la tabella di confusione è stata presa in considerazione una distinzione categorica fra valori "zero", considerando tutti i valori al di sotto di una soglia di 10^{-4} e "nonzero", considerando i valori al di sopra di tale soglia. Questi valori vengono considerati per i dati differenziali, ovvero valutando la differenza di valore di ogni timestep rispetto al precedente.

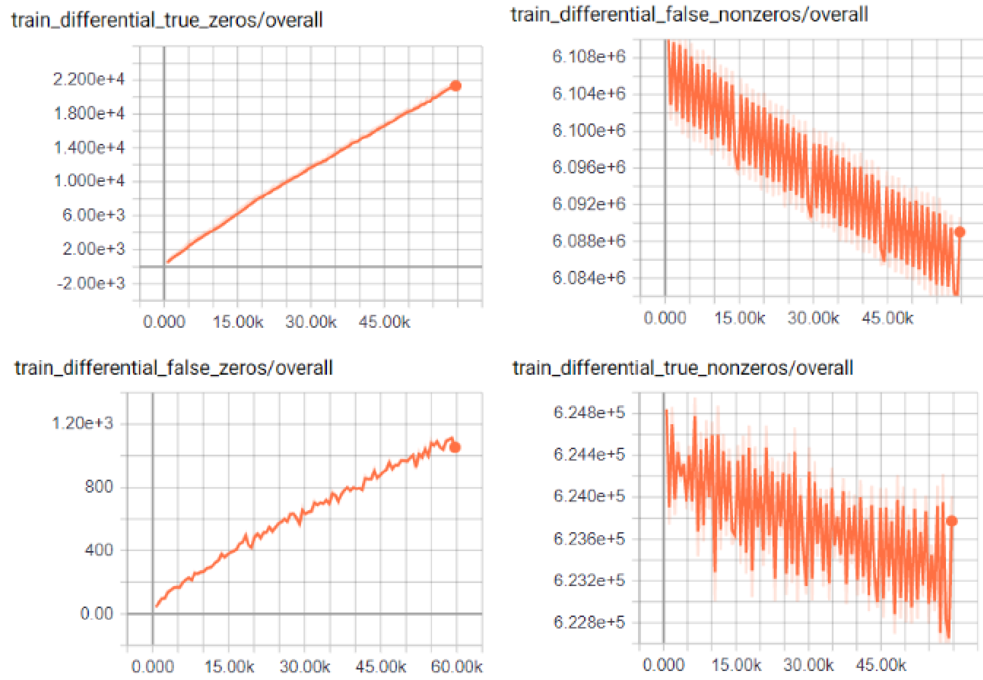


Figura 3.21: Tabella di confusione dai risultati del training "direct" su Tensorboard. In alto a sinistra i zeri corretti, in basso a sinistra i zeri scorretti, in alto a destra i valori maggiori di zero scorretti, in basso a destra quelli corretti.

Si hanno dei valori significativi sui "false non zero": questo significa che vengono predette frequentemente variazioni sui segnali, quando in realtà non ci sono. Ci si aspetta di avere un modello più tendente al "caotico" al contrario di un persistence model, che non prevede nessuna variazione. Osservando il training set, la rete esegue correttamente le previsioni, con un certo margine di tolleranza:

Figura 3.22: Esempio di predizione dal modello dopo il training "direct" con dataset "pioggia", su dati del training set. Si tratta dello stesso periodo preso in considerazione nella figura 3.18. Al contrario dell'altro modello, vengono previsti subito i rialzi dei livelli in seguito all'arrivo delle precipitazioni, per tutti i segnali critici in considerazione.

Le previsioni rispetto all'eval set evidenziano maggiormente il comportamento caotico intuito osservando le metriche. Infatti, soprattutto durante i fenomeni di pioggia, le previsioni tendono ad evidenziare una costante crescita o decrescita dei livelli. È importante sottolineare che il valore medio di queste previsioni si scosta poco rispetto al valore atteso, insieme al fatto che non si tratta di un persistence model.

Figura 3.23: Esempio di predizione dal modello dopo il training "direct" con dataset "pioggia", su dati dell'eval set. Rispetto al training set l'andamento è più caotico, ma prevede correttamente l'innalzamento dei livelli.

Training feedback

Il Mean Absolute Error generale del training differenziale presenta valori inferiori rispetto al training "diretto". Una prima impressione potrebbe essere di avere una rete più precisa rispetto all'altra.

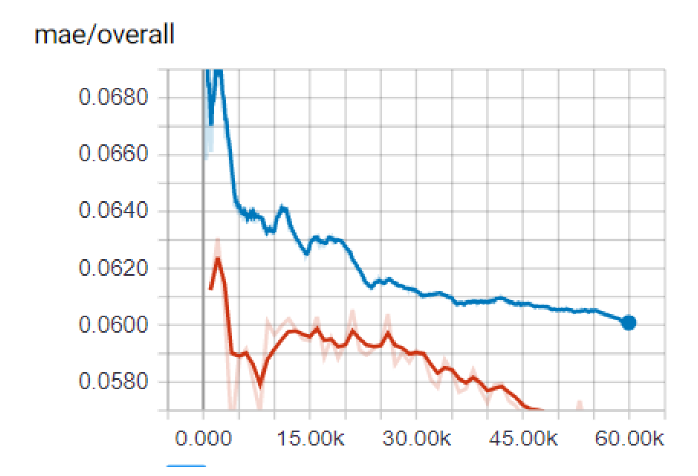


Figura 3.24: Risultati del MAE su Tensorboard per il training "feedback". In blu il training, in arancione l'evaluation. In questo caso sembrerebbe che l'evaluation stia avendo valori ottimi, ma verranno visti subito dopo i risultati effettivi.

Osservando la particolare matrice di confusione "zeros/nonzeros" si nota che il numero di "false zeros" è un ordine di grandezza superiore rispetto all'altra rete, tendendo ad aumentare. Anche il numero di "true zeros" è di un ordine di grandezza superiore.

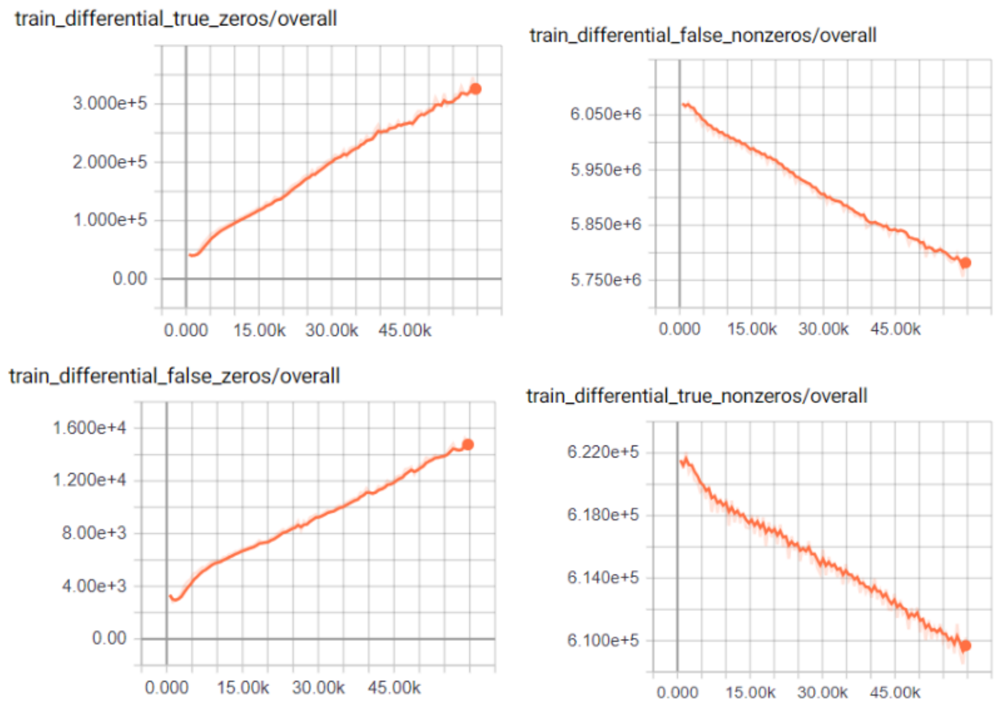


Figura 3.25: Tabella di confusione dai risultati del training "feedback" su Tensorboard. Un numero inferiore di segnali erroneamente maggiori allo zero sembrerebbe suggerire una previsione più precisa.

Andando ad osservare l'output previsionale della rete, si nota di avere di nuovo ottenuto un persistence model: ogni 20 minuti si ha uno "scatto" del livello, che si allinea all'ultima lettura ricevuta, e persiste per i 20 minuti successivi, ovvero i minuti di previsione. Di conseguenza, la metrica MAE espone la correttezza generale dei segnali in uscita dalla rete, senza tenere conto della bontà delle sole previsioni.

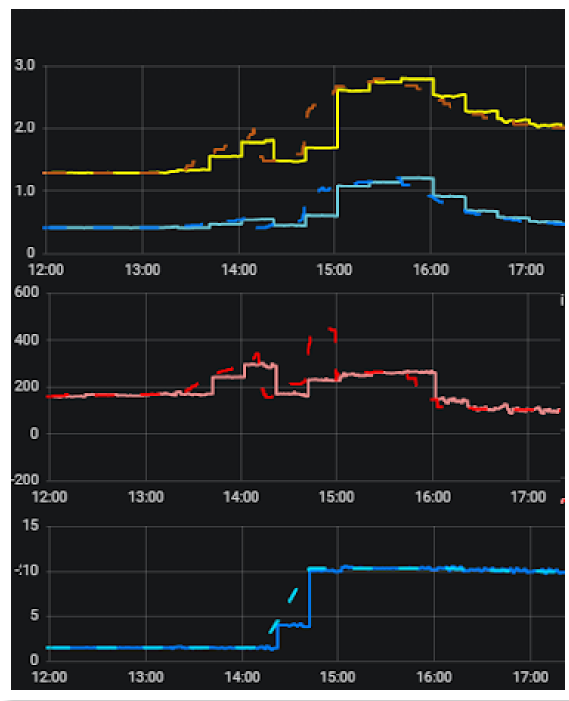


Figura 3.26: Esempio di previsione tipo "persistence model" con il training "feedback". Si ha nuovamente un andamento simile a quello visto su figura 3.18 sull'evaluation set.

Capitolo 4

Pipeline in streaming

In fase di produzione, il sistema deve essere in grado di fornire previsioni aggiornate con gli ultimi dati provenienti da SCADA. A tal fine, è necessario lavorare con informazioni in tempo reale, in modo di realizzare con periodicità frequente le predizioni con il sistema realizzato. Nelle pipeline in batch, l'intero insieme dei dati da elaborare è a disposizione al momento di avvio della pipeline. Su quelle in streaming, invece, i dati arrivano in tempo reale e il flusso deve eseguire le manipolazioni su questi dati mano a mano che vengono messi a disposizione. Esistono tutta una serie di meccanismi per gestire questo requisito in maniera scalabile. Dopo aver visto nel dettaglio questi strumenti e meccanismi, si andrà a studiare il caso d'uso affrontato dal candidato e la soluzione applicata. Per tutta questa sezione si può fare riferimento al libro Streaming Systems di Akidau, Chernyak e Lax [24].

4.1 Windowing

Il Windowing è una tecnica di suddivisione dei dati sulla base del loro timestamp. Il timestamp rappresenta l'istante temporale in cui una informazione è stata generata. Nel caso in studio si hanno delle letture di sensori, ogni lettura avrà un proprio timestamp. Una caratteristica di base del Windowing è che la suddivisione viene effettuata solo in seguito all'applicazione di una GroupByKey (vedi 3.2.2: GroupByKey). Questo perché il processo di Windowing è stato pensato per il calcolo distribuito, secondo i principi

base di Hadoop MapReduce, che ha la base la struttura $< chiave, valore >$ I dati quindi saranno divisi sia per chiave, sia per finestra temporale.

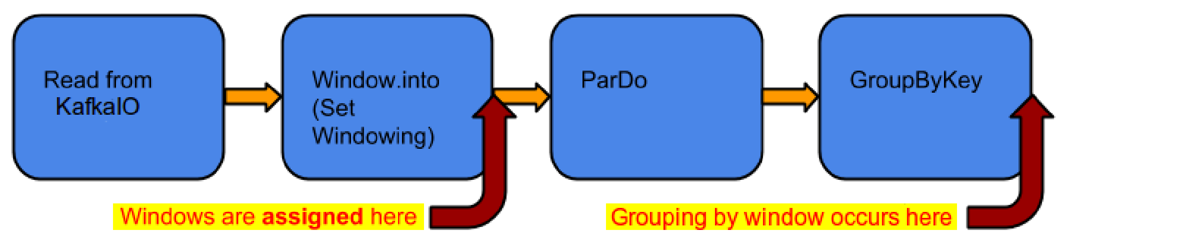


Figura 4.1: Processo di windowing [20]. Le finestre sono assegnate nel processo di Windowing, tuttavia la suddivisione vera e propria avviene solo quando viene effettuata anche la GroupByKey. Questo significa che qualunque operazione sia compiuta dalla ParDo raffigurata nel terzo blocco sarà eseguita indistintamente dalle finestre e dalle chiavi.

Esistono attualmente tre diversi tipi di windowing, che vanno a dividere in maniera differente le sequenze temporali.

4.1.1 Fixed

Nelle Fixed Windows le finestre hanno larghezza fissa e non si sovrappongono, con la stessa larghezza in tutte le chiavi degli elementi. Dopo la GroupByKey avremo delle PCollection di elementi raggruppati per chiave e per finestra. Nella prossima figura si vede una rappresentazione di questo meccanismo: si supponga di avere 3 chiavi, ognuna con i propri valori associati, con timestamp che vanno da 0 a 60 secondi. Si vogliono raggruppare gli elementi, oltre che per chiave, anche per finestre di grandezza fissa di 15 secondi. Si avranno quindi quindi quattro finestre consecutive di larghezza 15 secondi, ognuna delle quali conterrà gli elementi raggruppati per chiave e per intervallo temporale.

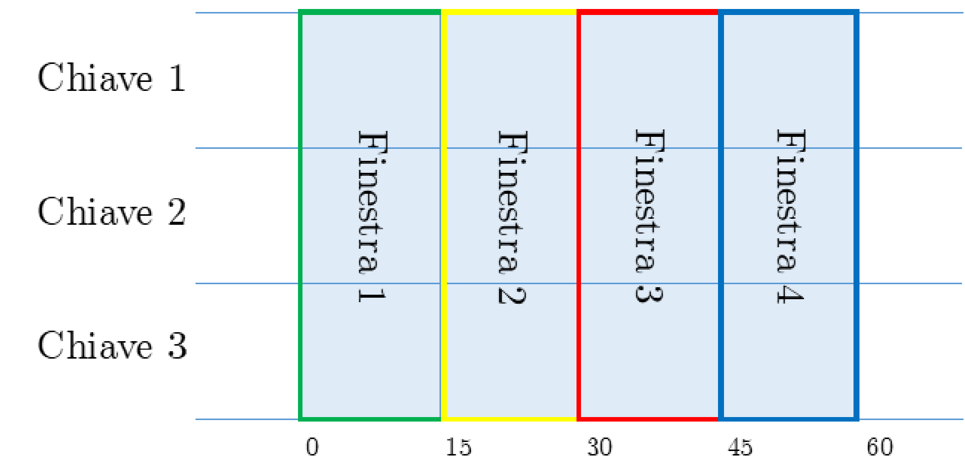


Figura 4.2: Fixed Windows. Ogni finestra ha grandezza fissa di 15 secondi.

4.1.2 Sliding

Le Sliding Window sono finestre mobili sovrapponibili, hanno stessa larghezza in tutte le chiavi degli elementi e si spostano di un'ampiezza fissa. Sono caratterizzate dal *periodo* e dallo *stride*: il periodo determina la larghezza della finestra, lo stride determina lo spostamento della finestra rispetto all'istante di partenza. Di conseguenza si avranno molte finestre sovrapposte sulla base di larghezza e stride delle finestre. Nella prossima figura si può vedere un esempio in cui supponiamo di avere elementi con tre chiavi differenti e timestamp distribuiti fra 0 e 60 secondi. Si supponga di raggruppare gli elementi per chiave e in finestre mobili di larghezza 30 secondi con stride 15 secondi. Come risultato, dopo la GroupByKey, si avranno tre finestre:

1. La prima finestra inizia a 0 secondi e finisce a 30 secondi
2. La seconda finestra inizia a 15 secondi e finisce a 45 secondi
3. La terza finestra inizia a 30 secondi e finisce a 60 secondi.

Gli elementi con timestamp fra 15 e 30 secondi finiranno sia nella prima che nella seconda finestra, similmente gli elementi con timestamp fra 30 e 45 secondi finiranno sia nella seconda che nella terza finestra.

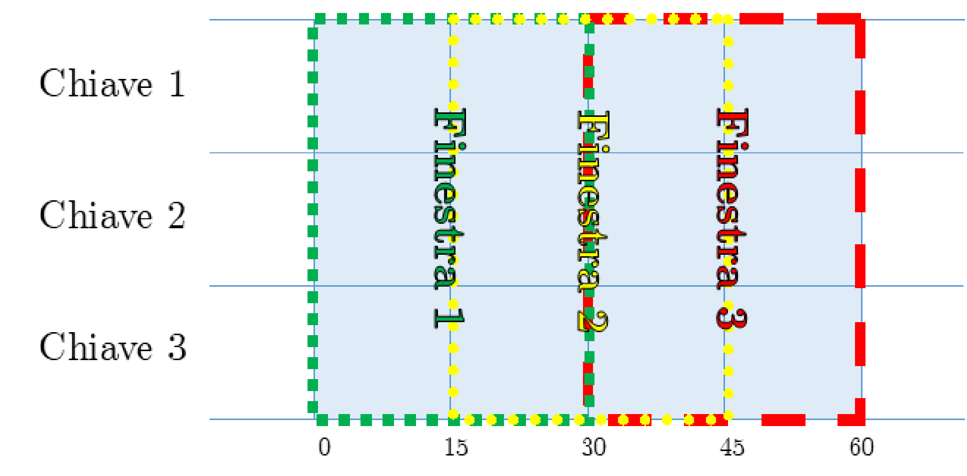


Figura 4.3: Sliding Windows. Ogni finestra ha larghezza fissa di 30 secondi, le finestre successive alla prima si spostano in avanti di 15 secondi rispetto la precedente e possono sovrapporsi.

4.1.3 Sessions

Le Session Window sono finestre di cui la larghezza non è determinata a priori, ma va stabilito un minimo intervallo fra una finestra e quella successiva, questo intervallo viene chiamato *gap duration*. Inoltre, la larghezza delle finestre può differire fra le chiavi degli elementi raggruppati, ovvero le finestre non sono allineate fra le diverse chiavi. L'obiettivo di questo tipo di finestra è raggruppare eventi adiacenti e separare eventi che avvengono in momenti differiti. Nella prossima figura si ha un esempio dove gli elementi hanno tre chiavi differenti e timestamp che vanno dai 0 ai 3 minuti e 45 secondi, ma di cui solo in certi intervalli sono presenti elementi, in particolare:

1. Per la chiave 1 si hanno elementi negli intervalli

   Da 0 a 45 secondi

   Da 1:30 a 1:45

   Da 2:00 a 2:15

   Da 3:00 a 3:45

2. Per la chiave 2 ci sono elementi negli intervalli

Da 0 a 1:00

Da 1:45 a 2:14

Da 2:30 a 3:00

3. Per la chiave 3 si hanno elementi negli intervalli

   Da 0:15 a 1:00

   Da 1:15 a 2:00

   Da 2:30 a 3:15

Si suppone di raggruppare gli elementi in Session Windows con gap duration di 30 secondi. Si avranno come risultato delle finestre diverse per ogni chiave, dal momento che i timestamp degli elementi sono distribuiti diversamente per ogni chiave. In particolare:

1. Per Chiave 1

   Una finestra da 0 a 45 secondi

   Una seconda finestra da 1:30 a 2:15, perché l'intervallo fra i dati è inferiore ai 30 secondi

   Una terza finestra da 3:00 a 3:45

2. Per Chiave 2

   Una finestra da 0 a 1:00

   Una seconda finestra da 1:45 a 3:00, perché l'intervallo intermedio è inferiore ai 30 secondi

3. Per Chiave 3

   Una finestra da 15 a 2:00, perché l'intervallo intermedio è inferiore ai 30 secondi

   Una seconda finestra da 2:30 a 3:15.

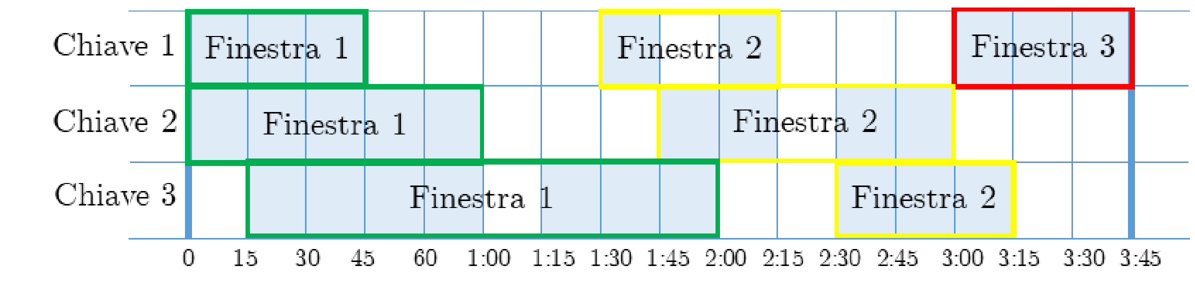


Figura 4.4: Session Windows. L'unico vincolo è la *gap duration* minima di 30 secondi rispetto alla finestra precedente. Le finestre possono sovrapporsi e sono indipendenti rispetto ogni chiave.

4.2 Publish-Subscribe

Nell'architettura di riferimento la sorgente dei dati è data da un sistema di Publish-subscribe [25]. In questo sistema si hanno i *publisher*, che pubblicano eventi strutturati e i *subscriber* che si sottoscrivono agli eventi. Dal momento che ogni subscriber può essere interessato solo a un sottoinsieme degli eventi pubblicati dai publisher, esistono dei sistemi di filtraggio. Su Google PubSub si ha un sistema Topic-based (o subject-based). Con questo approccio, si pone l'assunzione che ogni notifica sia espressa in un formato di diversi campi di cui campo fisso è il topic. Un subscriber si può abbonare a uno o più topic, quindi riceverà gli eventi relativi solamente a quei topic. Nella prossima figura si vede un esempio con tre publisher A, B, C e tre subscriber X, Y, Z. I tre publisher pubblicano sui rispettivi topic A, B, C un messaggio. Subscriber X è iscritto, tramite le Subscription XA e XB, ai topic A e B . $SubscriberY$ è iscritto al topic C tramite la subscription YC, anche Subscriber Z è iscritto al topic C tramite la subscription ZC. I tre publisher inviano rispettivamente tre messaggi $Message1, Message2, Message3$. $Message1$ e $Message2$ verranno inviati a $SubscriberX$, mentre $Message3$ sarà inviato ai subscriber Y e Z.

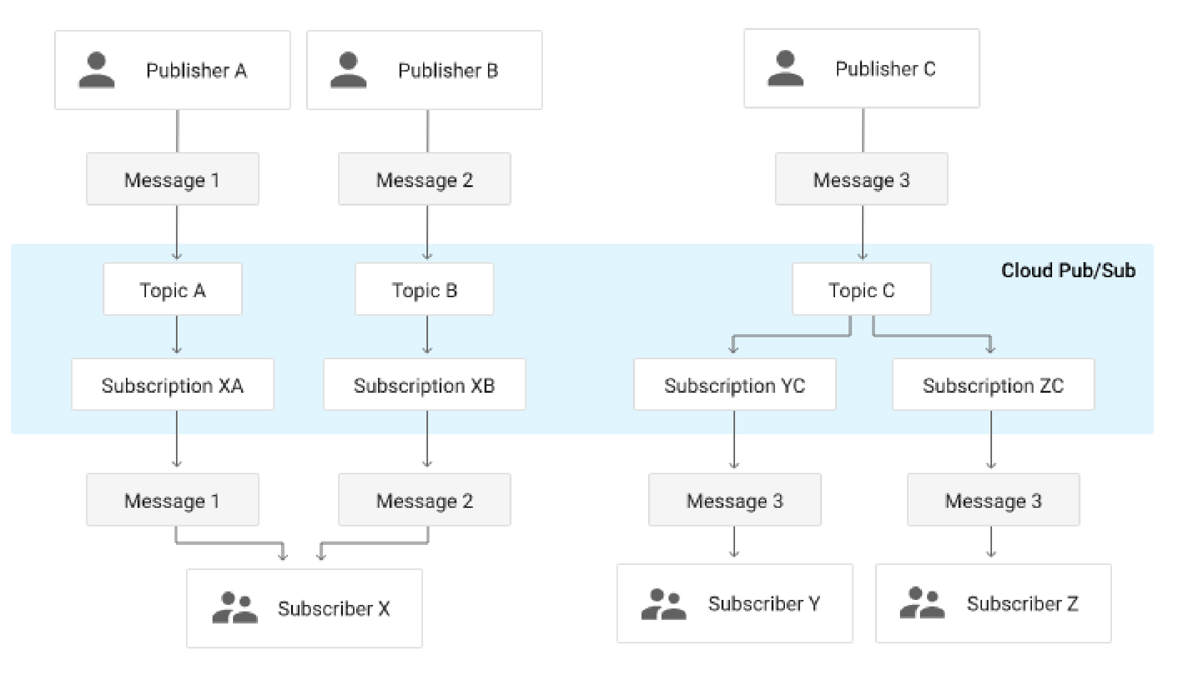


Figura 4.5: Esempio di rete Pub/Sub. A, B e C sono publisher, X, Y e Z sono subscriber, A, B e C sono topic.

Un sistema di Publish-Subscribe ha le seguenti caratteristiche, di cui è necessario tener conto nel momento in cui si va a realizzare una architettura che manipola i dati in tempo reale:

- Eterogeneità: I componenti che usano il sistema Publish-Subscribe possono essere completamente differenti, sfruttando questo sistema di comunicazione per potersi interfacciare e diventare interoperabili. Nel caso in analisi, infatti, i dati saranno provenienti da uno SCADA, verranno invece elaborati da una pipeline realizzata con il framework Beam in esecuzione su Google Dataflow.
- Asincronicità: Le notifiche ai Subscriber sono inviate in maniera asincrona rispetto ai Publisher. Questo significa che al momento della pubblicazione di un evento, il Publisher non dovrà attendere che il messaggio sia inviato a tutti i Subscriber, ma il sistema provvederà a inviare la comunicazione a tutti gli iscritti.

Google Pub/Sub, inoltre, ha le seguenti caratteristiche:

- Nessuna garanzia di ordine: i messaggi vengono inviati con la massima velocità possibile, ma con nessuna garanzia di ordine.
- Garanzia di at-least-once: i messaggi vengono spediti ai subscriber almeno una volta, tuttavia potrebbero esserci dei messaggi duplicati anche più di una volta.

La pipeline in streaming potrà andare a leggere direttamente gli eventi provenienti da un sistema Publish-Subscribe di questo tipo, iscrivendosi a un topic e ricevendo i dati in ingresso da quel topic, come vediamo nel seguente snippet di codice Java:

```
PCollection<Event> events = p.apply(eventType + "pubsub:read",
        PubsubIO.readStrings().fromTopic(topic)
        .withTimestampAttribute("EventTimeStamp")
        ) //
```

Figura 4.6: Codice Java di lettura da un topic PubSub. *p* è la pipeline, vengono letti oggetti Event da un topic di PubSub, conservando il timestamp originale di quegli eventi.

Sarà importante quindi tenere conto delle caratteristiche di questo sistema di comunicazione per l'elaborazione dei dati richiesta.

4.3 Watermark

Nelle pipeline in streaming, come quelle che leggono i dati provenienti da PubSub con l'iscrizione a un topic, ci sono una serie di concetti aggiuntivi rispetto alle pipeline di base. Questi concetti sono relativi ai vincoli temporali che vengono dati per l'elaborazione di questi dati. Per questo motivo, il timestamp è una informazione indispensabile e necessaria per qualunque sorgente di dato. Se il esso non viene fornito dalla sorgente, sarà utilizzato l'istante in cui il dato è entrato a disposizione della pipeline. Ad esempio, supponiamo di avere come sorgente dati un lettore PubSub iscritto a un topic. Per ogni messaggio, se non viene letto il timestamp da un suo attributo, sarà utilizzato l'istante in cui il messaggio è ricevuto dalla pipeline. Un primo vincolo generale che si applica alla sorgente dati è il Watermark. Qualunque informazione in arrivo avente un timestamp successivo al Watermark verrà trascurata e non verrà mai elaborata dalla pi-

peline. Questo concetto è necessario al fine di poter garantire il funzionamento corretto dell'operazione di Windowing che abbiamo visto in precedenza, la quale opera sui timestamp. Il Watermark rappresenta lo stato di completamento dell'elaborazione rispetto a un certo istante. Nel grafico che segue vediamo l'interazione del Watermark con gli altri aspetti temporali dei dati in streaming. Sull'asse orizzontale abbiamo *Event Time*, che caratterizza il timestamp del dato. Questo può essere, ad esempio, l'istante di lettura da un sensore. Sull'asse verticale abbiamo il *Processing Time*, ovvero l'istante in cui lo stesso dato viene elaborato dalla pipeline. Incrociando i due istanti, otteniamo un punto del grafico. Il caso ideale, impossibile in ambito reale, è rappresentato dalla bisettrice del piano, dove l'istante in cui il dato viene prodotto ha un ritardo ben determinato rispetto al momento in cui viene processato. In realtà si ha un ritardo variabile nel tempo, sul grafico indicato come *Processing time lag*, da cui otteniamo l'andamento rappresentato in rosso.

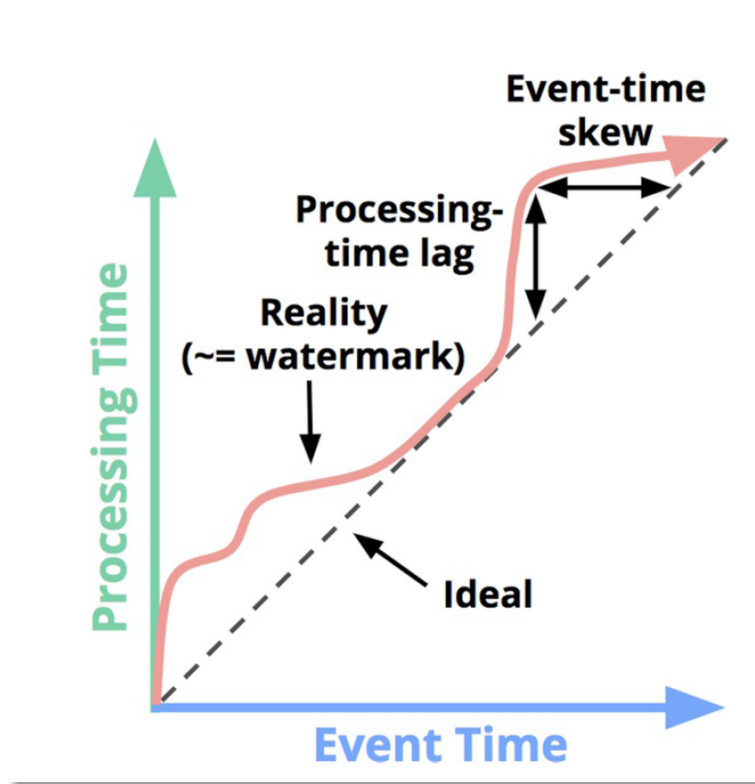


Figura 4.7: Watermark, processing time, event time [24]. L'asse orizzontale è il timestamp dell'evento, l'asse verticale è l'istante in cui lo stesso evento viene processato dalla pipeline. La funzione rossa rappresenta l'andamento del ritardo di processamento, che non è costante.

4.4 Triggering

Nelle implementazioni reali dei sistemi in streaming come Beam, il Watermark viene determinato in maniera euristica, sulla base delle caratteristiche dei dati che provengono in streaming, per compensare sorgenti che non garantiscono la delivery in ordine come Pub/Sub. Il ritardo di processamento dei dati non è deterministico, di conseguenza il Watermark euristico potrebbe non essere desiderabile. Si potrebbero volere invece i dati in uscita da una pipeline entro un certo ritardo massimo, eventualmente trascurando i dati che arrivano in ritardo eccessivo. Questo è possibile utilizzando i Trigger in combinazione al Windowing. In questo modo, è possibile raggruppare i dati all'interno di finestre temporali e stabilire quando una certa finestra avrà esaurito il proprio contenuto.

Esistono molti tipi di trigger, ognuno si comporta in maniera differente. Essi possono essere utilizzati in combinazione, per ottenere il comportamento desiderato. Tutti hanno la caratteristica di determinare il momento in cui i dati all'interno di una finestra verranno resi disponibili per l'elaborazione al resto della pipeline, dopo l'esecuzione della GroupByKey. Questo momento viene denominato *firing*, a significare che il contenuto di quella finestra recuperato fino a quell'istante viene "sparato" fuori dalla GroupByKey. L'autore del libro Streaming Systems, Tyler Akidau, fa riferimento ai vari aspetti combinazione delle Window e dei Trigger come 4 elementi principali: cosa (What), dove (Where), come (How), e quando (When). Per quanto riguarda cosa (What), parliamo del raggruppamento e delle operazioni che saranno eseguite sui dati dopo di esso, nel framework Beam abbiamo visto le GroupByKey e le ParDo. Il dove (Where) viene stabilito dal Windowing, che stabilisce dove andare a suddividere temporalmente i nostri dati. Il come (How) e quando (When) saranno stabiliti dai tipi di trigger e dal loro comportamento, che adesso vedremo.

4.4.1 Comportamento del trigger

Gli elementi che rientrano in quella finestra potrebbero non essere finiti dopo il primo firing, quindi potremmo averne più di uno per la stessa finestra temporale. Il comportamento (How) determina cosa succede agli elementi della stessa finestra nel momento in cui avvengono i firing successivi al primo.

Accumulating

Per ogni firing, il contenuto delle finestre precedentemente emesse (*fired panes*) viene ripetuto. Per esaminare meglio questo comportamento, si prenda come riferimento un esempio: si supponga di ricevere eventi contenenti un numero intero e un proprio timestamp. Si vogliono dividere questi eventi in finestre fisse di larghezza 2 minuti e calcolare la somma del valore degli eventi per ciascuna finestra. Per il momento si trascuri la modalità in cui sono scattati i trigger, prendendo per noto che ad ogni somma parziale è scattato un trigger. Nella figura seguente si ha un comportamento Accumulating su questo esempio, dove si vede:

- sull'asse orizzontale, il timestamp dell'evento

- sull'asse verticale, il timestamp dell'istante di processamento
- nei cerchi grigi i singoli eventi con il proprio valore

  i singoli eventi hanno un proprio timestamp e un proprio istante di processamento
- nelle righe verticali la separazione in finestre

  le finestre sono di larghezza fissa di 2 minuti
- nelle righe orizzontali i trigger scattati (firing)

  per ogni firing, viene calcolata la somma parziale degli elementi al suo interno (numeri in giallo)
- nei numeri in grassetto: le somme parziali in giallo, quelle totali in bianco.

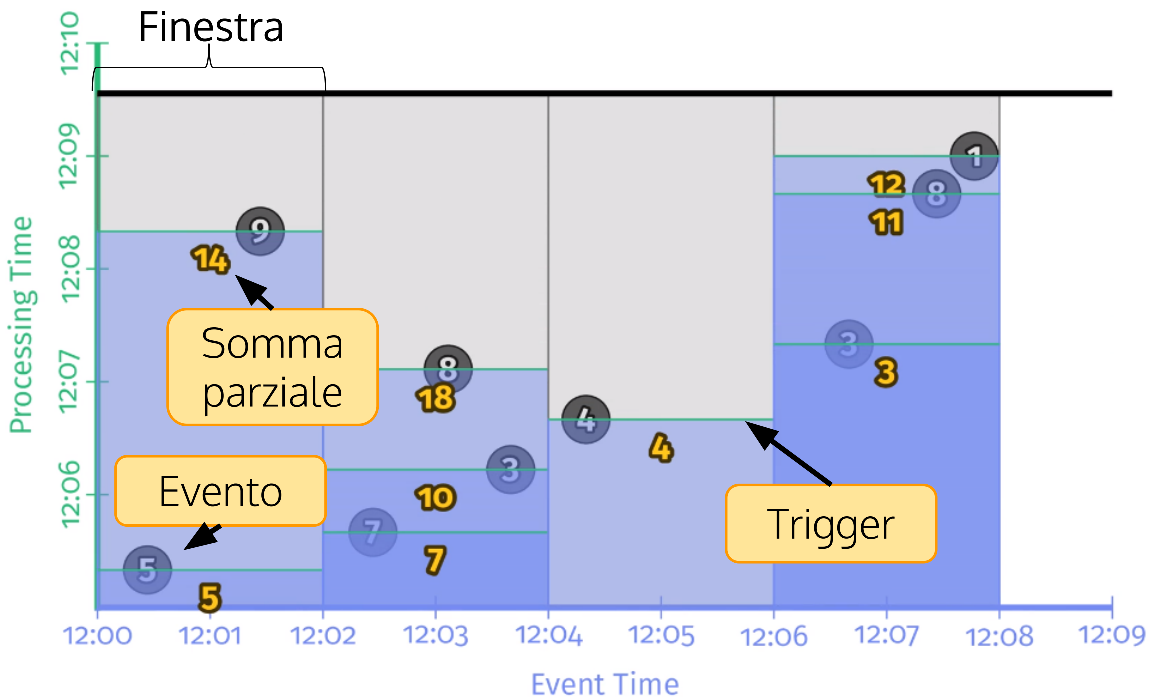


Figura 4.8: Accumulating fired panes. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Versione modificata di un video tratto da [26].

In questo caso, si ha che le somme parziali ottenute alla fine dei firing, ovvero l'ultima somma emessa per ciascuna finestra, corrispondono al risultato atteso. Questo perchè il comportamento Accumulating accumula gli eventi della stessa finestra. A ogni nuovo firing vengono mandati tutti gli elementi che rientrano nella stessa finestra, anche se erano già stati emessi.

Discarding

Per ogni firing vengono emessi solo i nuovi elementi, i precedenti non sono ripetuti. Si consideri l'esempio di prima, ma con modalità Discarding fired panes. In questo caso, per ottenere la somma complessiva in una finestra, si dovrebbe memorizzare il valore emesso ad ogni firing e sommarlo a quello ottenuto precedentemente.

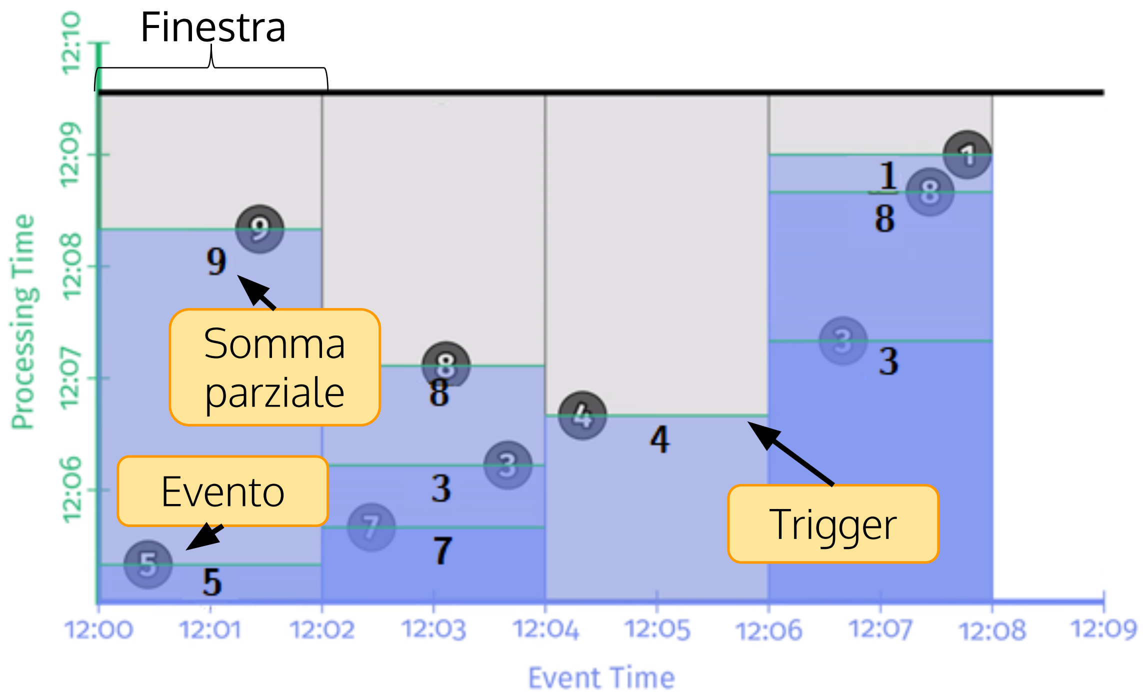


Figura 4.9: Discarding fired panes. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Versione modificata di un video tratto da [26].

Accumulating and retracting

Viene ripetuto l'intero contenuto della finestra, come per Accumulating, ma viene anche fornita la differenza rispetto ai valori accumulati fino a quel momento. Questa situazione è utile per quei casi in cui l'operazione di aggregazione ha bisogno di conoscere il cambiamento dell'insieme dei dati per calcolare il nuovo valore.

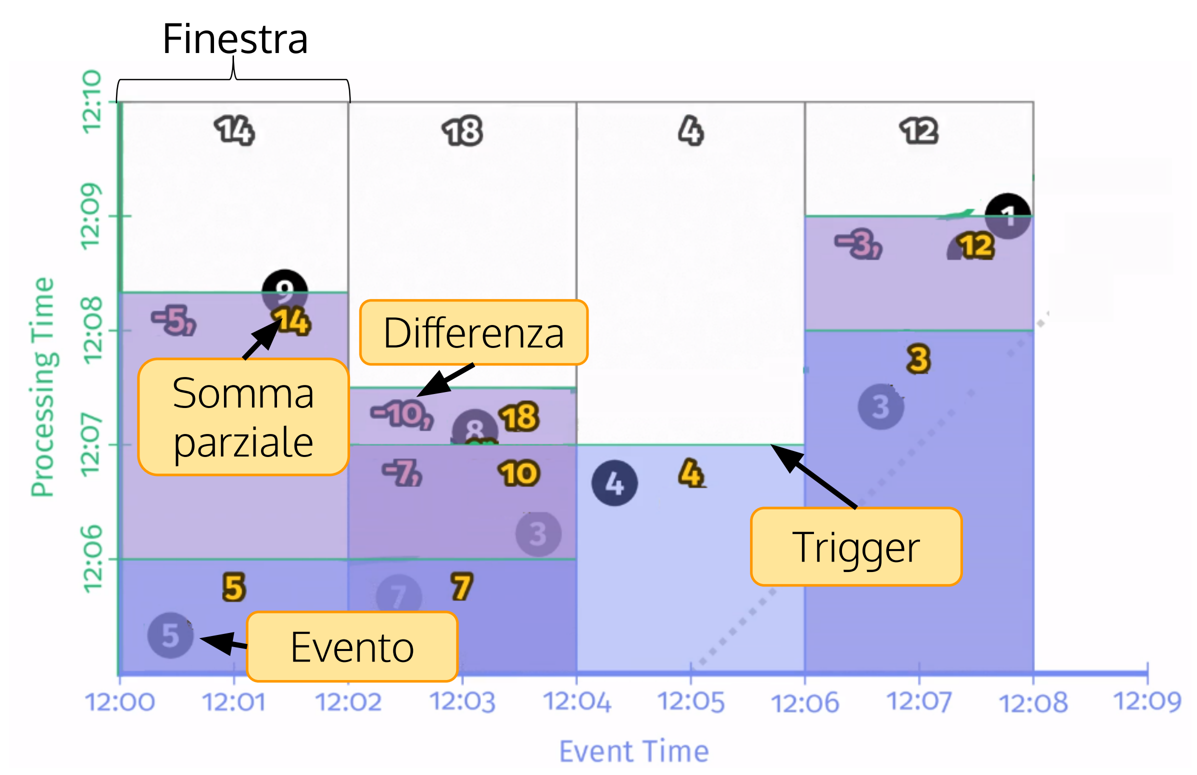


Figura 4.10: Accumulating and retracting fired panes. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale e la differenza. Versione modificata di un video tratto da [26].

In certi casi in cui vengono calcolati risultati parziali, potrebbe essere importante conoscere la differenza rispetto ai valori già ricevuti.

4.4.2 Tipologie di trigger

Il tipo di trigger determina quando (when) verranno processati i dati in ciascuna finestra. Esistono due macro-tipologie di trigger: *Repeated update* trigger e *Completeness* trigger. La differenza sostanziale è l'utilizzo o meno del watermark euristico.

Repeated update triggers

Questo tipo di trigger scatta indipendentemente dallo stato del watermark, quindi garantiscono un controllo più specifico nel caso in cui si vogliano seguire gli andamenti dei dati in arrivo, in particolare dal punto di vista della dimensionalità e del tempo. Con Repeated update si intende che scatta più volte, ogni qualvolta si verifica nuovamente la stessa condizione.

Per dimensionalità (per-record) I trigger basati sulla dimensionalità (per-record triggers) scattano ogni qualvolta si ha almeno un dato numero di elementi processati in una finestra, contando indipendentemente per ogni finestra tale numero. Questo permette di avere la certezza che i dati messi a disposizione avranno sempre una cardinalità minima. Prendendo in esame l'esempi visto prima, si ipotizzi di utilizzare finestre fisse di larghezza 2 minuti e un Repeated update trigger per dimensionalità minima 1:

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO MINUTES))
               .triggering(Repeatedly(AfterCount(1))));
  .apply(Sum.integersPerKey());
```

Figura 4.11: Pseudocodice da [24] per la definizione di un Per-record trigger.

Si avrà un firing ogni qualvolta viene processato un elemento che rientra in una finestra. Nella prossima figura si vede questo effetto, si notino le frecce che rappresentano lo scatenarsi del trigger:

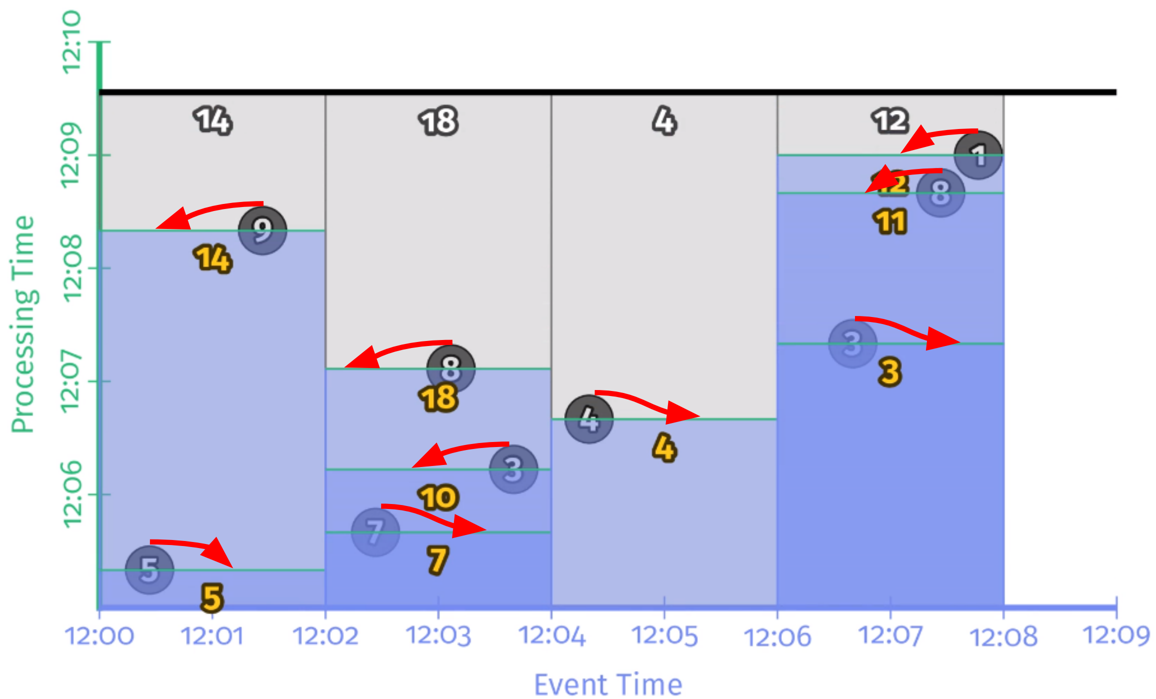


Figura 4.12: Esempio per-record triggers. Sull'asse orizzontale si ha il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Versione modificata di un video tratto da [26].

Le somme parziali seguono il comportamento Accumulating, per cui all'interno di ogni finestra si avrà l'emissione di una nuova somma parziale ogni qualvolta viene letto un nuovo valore nella stessa finestra. ***Per ritardo non allineato (unaligned delay)***

I trigger per ritardo disallineato (unaligned delay trigger) scattano dopo un certo ritardo temporale rispetto al primo elemento processato all'interno di una finestra, contando questo ritardo indipendentemente per ogni finestra. Il ritardo è determinato rispetto al tempo di processamento degli elementi, non rispetto al tempo di inizio o fine finestra. Nel caso in cui sia già scattato, si considera il primo elemento processato in una finestra successivamente al firing precedente. In questo modo, si avrà garanzia di avere un ritardo massimo limitato fra la disponibilità di un dato e il completamento della sua elaborazione. Si consideri questo tipo di trigger per l'esempio visto finora, con finestre di 2 minuti: si supponga di avere un unaligned delay di 2 minuti.

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(Repeatedly(UnalignedDelay(TWO_MINUTES))
  .apply(Sum.integersPerKey());
```

Figura 4.13: Pseudocodice da [24] per la definizione di un Unaligned delay trigger.

Si possono vedere 5 trigger:

1. Nella prima finestra che va dalle 12:00 alle 12:02, il primo elemento con timestamp 12:00:30 arriva alle 12:05:20, quindi il trigger scatta alle 12:07:20.
2. Sempre nella prima finestra, arriva un secondo elemento, con timestamp 12:01:30, alle 12:08:20. Dato che il primo trigger è già scattato si avvia un altro contatore, che fa scattare il trigger alle 12:10:20.
3. Nella seconda finestra dalle 12:02 alle 12:04, il primo elemento arriva alle 12:05:40, che avvia il contatore di 2 minuti. Arrivano altri 2 elementi, ma tutti prima delle 12:07:40, istante in cui scatta il trigger. In tale istante vengono emessi tutti gli elementi "catturati" fino a quel momento in tale finestra.
4. Nella terza finestra dalle 12:04 alle 12:06, arriva un solo elemento alle 12:06:30, quindi il trigger scatta alle 12:08:30.
5. Nella finestra dalle 12:06 alle 12:08 arriva un primo elemento alle 12:07:20. Questo elemento fa avviare il contatore del trigger, che scatta alle 12:09:20. Come con il terzo trigger che abbiamo esaminato, arrivano altri due elementi, entrambi prima dell'istante in cui il trigger scatta. Quindi alle 12:09:20 vengono emessi tutti gli elementi catturati fino a quell'istante nella finestra.

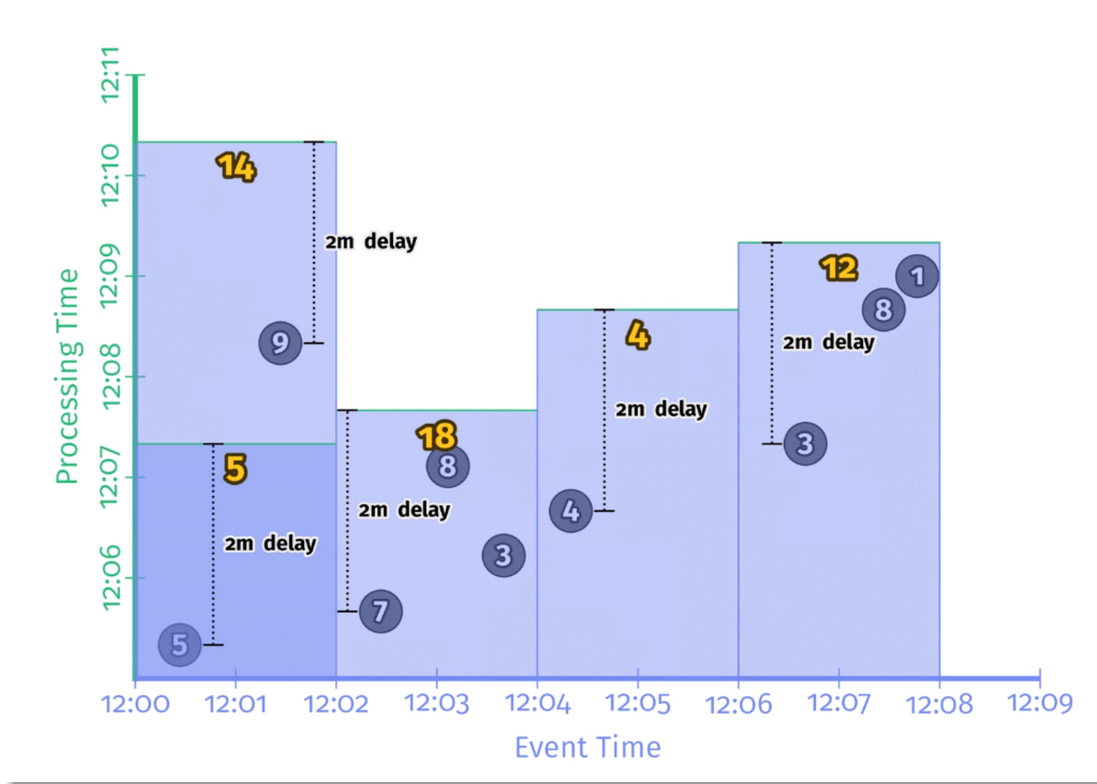


Figura 4.14: Esempio trigger unaligned delay [24]. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Per ogni contatore che fa scattare un trigger è mostrata in verticale una retta che evidenzia il tempo trascorso dall'evento all'istante in cui scatta. Versione modificata di un video tratto da [26].

Per ciascuno di questi trigger viene calcolata la somma parziale all'interno della stessa finestra, in modalità Accumulating.

Per ritardo allineato (aligned delay)

I trigger per ritardo allineato (aligned delay triggers) considerano sempre un ritardo predefinito rispetto agli elementi in arrivo, però tale ritardo è allineato fra tutte le finestre. Questo significa che il primo elemento in arrivo in una finestra fa avviare un timer, allo suo scadere scatenerà un firing per qualunque finestra, qualora vi siano elementi al suo interno. Questo tipo di trigger in particolare non è implementato su tutti i sistemi in streaming: in Beam non è implementato. Il requisito di sincronizzare il timer rispetto

tutte le finestre è caratteristica peculiare di questo tipo di trigger. Come esempio, si prenda sempre il caso visto in precedenza, supponendo di avere finestre fisse di 2 minuti con un Repeated update trigger a ritardo allineato di 2 minuti.

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(Repeatedly(UnalignedDelay(TWO_MINUTES))
  .apply(Sum.integersPerKey());
```

Figura 4.15: Pseudocodice da [24] per la definizione di un Aligned delay trigger.

Avremo lo scatenarsi di due trigger:

1. Il primo trigger è avviato dall'evento delle 12:05 e scatta dopo due minuti alle 12:07. Fra le 12:05 e le 12:07 sono arrivati elementi in altre finestre: vengono anch'essi emessi alle 12:07.
2. Il secondo trigger è avviato dall'evento delle 12:07 e scatta alle 12:09. Anche in questo caso, sono arrivati elementi in quel lasso di tempo in altre finestre: allo scattare del trigger verranno emessi anche gli elementi di quelle finestre.

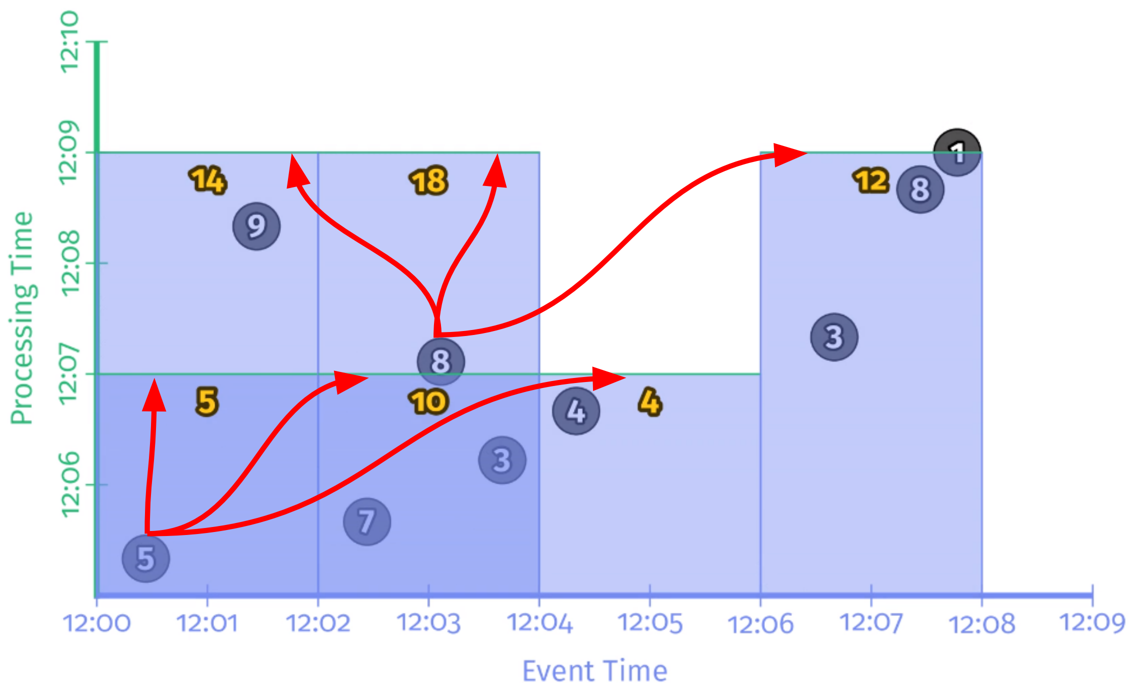


Figura 4.16: Esempio trigger aligned delay. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Per ogni evento che fa scattare un trigger sono visibili delle frecce rosse che evidenziano l'istante in cui è scattato. Essendo allineato, il firing avviene per tutte lefinestre. Versione modificata di un video tratto da [26].

Le somme parziali all'interno della stessa finestra sono sempre calcolate in modalità Accumulating.

Completeness trigger

Questo tipo di trigger segue l'andamento del Watermark, combinandolo con i Repeated Update trigger. In questo modo viene fatta distinzione fra elementi in "orario", in "anticipo" o in "ritardo" rispetto al watermark. Vediamo prima il caso in "orario", dove il trigger di ogni finestra scatta una volta soltanto, quando il watermark raggiunge il timestamp di fine finestra. Si supponga il caso di esempio già visto, con windowing su finestre fisse di 2 minuti, con l'uso solamente di un Completeness Trigger On Time.

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(AfterWatermark()))
  .apply(Sum.integersPerKey());
```

Figura 4.17: Pseudocodice da [24] per la definizione di un On-time completeness trigger

Il watermark euristico, di cui possiamo seguire l'andamento nella linea verde, cercherà di seguire l'andamento dei dati. Si ha il caso di un evento in "ritardo":

- L'evento perso ha timestamp 12:01:40
- Alle 12:06 il watermark ha valore 12:02, quindi considera la finestra chiusa e scatta il trigger "on time" per quella finestra
- Alle 12:08:20 arriva un nuovo evento (con valore 9, in rosso nella figura), che viene perso perchè nessun altro trigger è impostato su quella finestra.

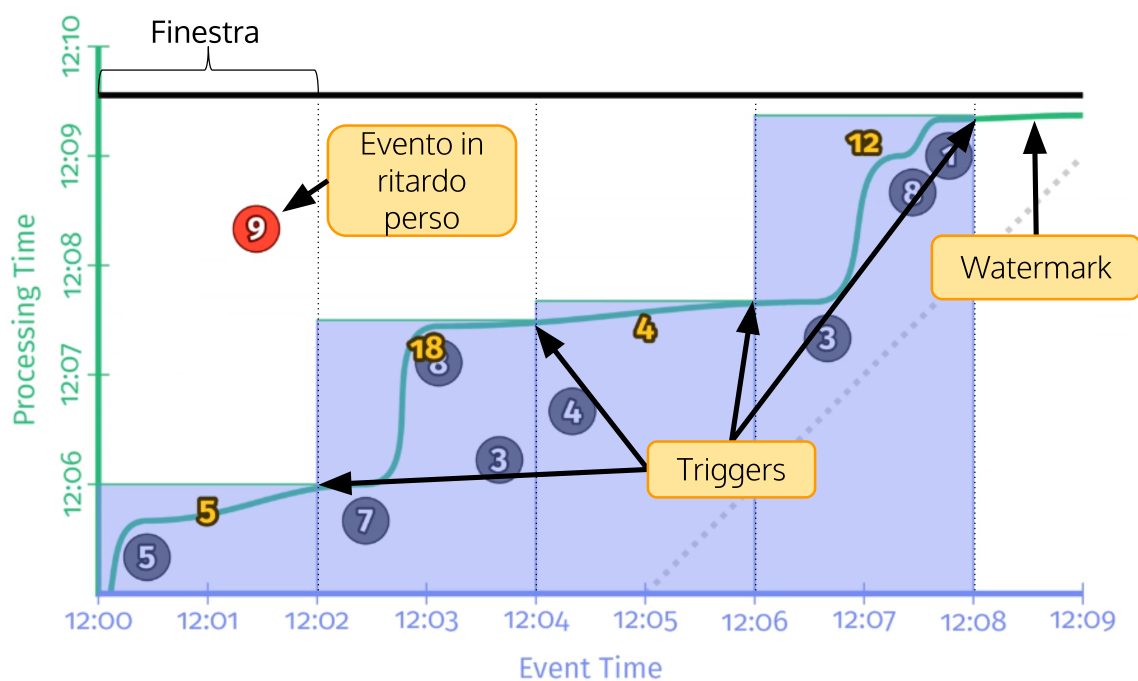


Figura 4.18: Esempio On Time trigger. Sull'asse orizzontale si vede il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Nell'istante in cui il watermark incontra l'istante di chiusura finestra, scatta il trigger per quella finestra. Un evento arriva con ritardo eccessivo e viene perso. Versione modificata di un video tratto da [26].

Il risultato è che la somma dei valori degli elementi della prima finestra, che va dalle 12:00 alle 12:02, è sbagliata, perchè il secondo elemento con valore 9 è stato perso.

Allowed lateness

Si è visto che i dati in arrivo dopo il watermark vengono scartati, di conseguenza non dovrebbe essere possibile avere dei trigger che scattano dopo il watermark. Questo è possibile, invece, con l'introduzione del "ritardo massimo consentito" (*allowed lateness*), noto fra i Streaming Systems anche come *Garbage Collection*. Questa misura determina il ritardo massimo tollerato dal sistema, dopo questo ritardo massimo rispetto al Watermark non sarà possibile avere alcun *late firing*.

Si hanno quindi tre tipi di Completeness Triggers, che possono essere gestiti separatamente:

- **Early firings**: essi vanno caratterizzati con uno o più Repeated Update Trigger, scattano qualora si verifichino le loro condizioni prima del Watermark.
- **On time**: quando il Watermark raggiunge il timestamp di chiusura finestra, scatta questo trigger. I dati che arrivano dopo che questo esso è scattato verranno considerati in ritardo.
- **Late firings**: questi trigger, come gli Early Firings, vanno caratterizzati con uno o più Repeated Update Trigger. I timer o contatori di tali trigger verranno attivati solo dopo che il Watermark ha fatto scattare il trigger "On time".

Si supponga di avere l'esempio visto in precedenza, ma oltre al trigger On Time aggiungiamo altri due trigger:

- Un Early Firing trigger costituito da un Aligned Delay di un minuto
- un Late Firing trigger costituito da un Per-Record trigger a un elemento.

```
PCollection<KV<Team, Integer>> totals = input
  .apply(Window.into(FixedWindows.of(TWO_MINUTES))
               .triggering(AfterWatermark()
                            .withEarlyFirings(AlignedDelay(ONE_MINUTE))
                            .withLateFirings(AfterCount(1))))
  .apply(Sum.integersPerKey());
```

Figura 4.19: Pseudocodice da [24] per la definizione di tre trigger, rispettivamente Early, On-Time e Late.

Nell'esempio in analisi avremo 8 firing, in particolare:

- I firing on-time si hanno quando il watermark raggiunge l'istante di fine finestra e in quel momento la stessa finestra contiene uno o più elementi
- Il late firing (nell'esempio è solo uno) accade nel momento in cui almeno un elemento rientra in una finestra, ma per quella finestra è già scattato il trigger on-time

- Gli early firing avvengono quando entra almeno un elemento in una qualunque finestra ed entro un minuto non scatta il trigger On-time. Essendo Repeating Trigger, possono scattare anche più di una volta prima dell'On-time trigger.

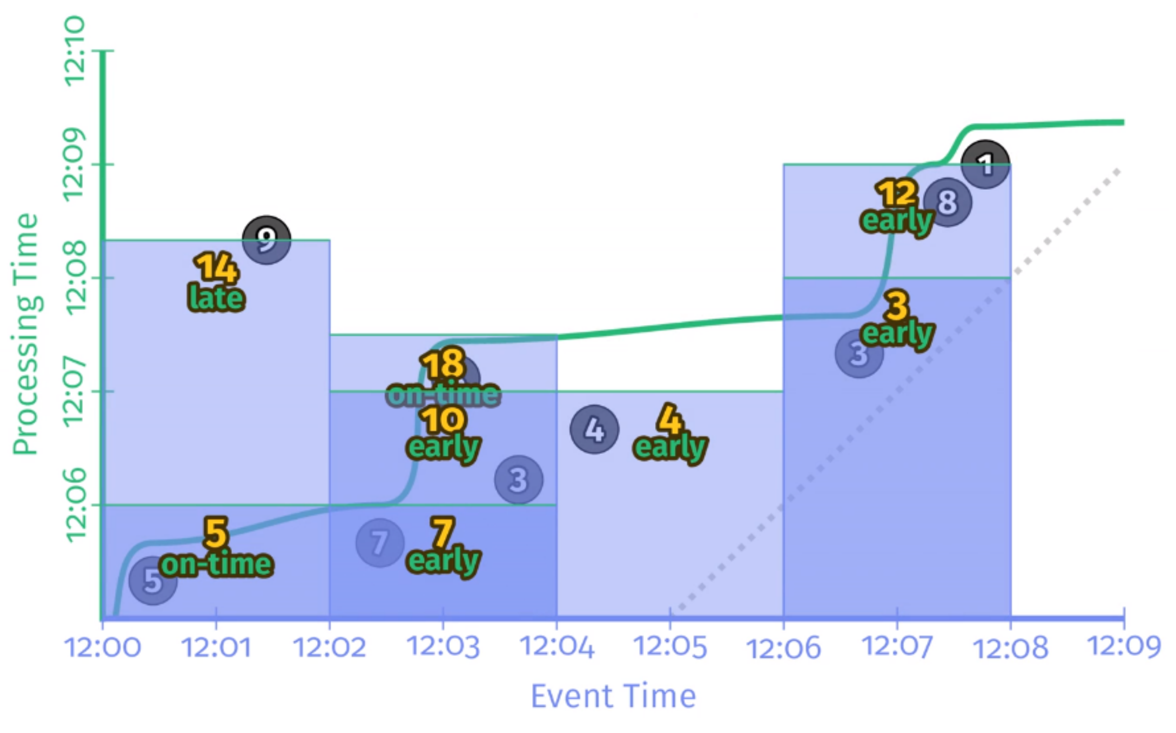


Figura 4.20: Esempio Early, On Time, Late triggers. Sull'asse orizzontale si ha il timestamp dell'evento, su quello verticale l'istante in cui viene processato dalla pipeline. Gli eventi sono divisi per finestre di larghezza fissa pari a 2 minuti. Ogni evento ha un certo valore intero, per ogni trigger è calcolata la somma parziale. Nell'istante in cui il watermark incontra l'istante di chiusura finestra, scatta il trigger On-time per quella finestra. Se le condizioni del trigger Early avvengono prima del trigger On-time, essi scattano. Similmente per i Late, se le condizioni avvengono dopo On-time. Versione modificata di un video tratto da [26].

In questo capitolo si è vista una panoramica dei concetti e degli strumenti principali richiesti per realizzare una architettura di processamento dei dati in streaming. Nel prossimo si vedrà il caso d'uso specifico affrontato dal candidato, provando diverse combinazioni di questi strumenti per ottenere i risultati e le prestazioni desiderate.

Capitolo 5

Sample and Hold in Streaming

È stato visto nella sezione 2.2.1 il problema del campionamento dei segnali, che richiede di avere un campione per tutti i valori delle telemetrie ogni 15 secondi, per poi aggregare i valori e ottenere campioni a 1 minuto. Questi ultimi tengono conto correttamente dell'andamento di ciascun segnale e vengono utilizzati come input della rete neurale, dove ogni minuto rappresenta un timestep. Nella fase di realizzazione del dataset di training si avevano a disposizione tutte le telemetrie e i dati meteo necessari, per ottenere i campioni ogni 15 secondi era sufficente eseguire la query InfluxDB vista su 2.6. Una volta preparata la rete per ottenere le previsioni, la si può utilizzare con i segnali e il meteo in tempo reale. Un primo problema affrontato è quello del campionamento con la tecnica Sample and Hold supponendo di avere le letture segnali in streaming. Si potrebbe ipotizzare di scrivere direttamente le letture su InfluxDB con una ingestione periodica, che in effetti funzionerebbe, avendo però i seguenti limiti:

- Alcuni segnali non inviano alcun dato nuovo per svariate ore, dato che non cambiano
- L'operazione di Sample and Hold con InfluxDB andrebbe eseguita su un intervallo di ore, dando per certo che in esso tutti i segnali abbiano avuto almeno una lettura

    Ogni richiesta è onerosa in termini di tempo

    Non scalerebbe bene su un numero maggiore di sensori

InfluxDB non è affidabile sulle query troppo impegnative e potrebbe smettere di rispondere per un tempo indefinito

- Tale operazione andrebbe ripetuta ogni 20 minuti

Per questi motivi si è ritenuto indispensabile realizzare una architettura più affidabile e veloce, studiata appositamente per trasformare i dati in tempo reale. Le letture dei sensori e del meteo vengono mandati su Google PubSub e sono processati da una pipeline in streaming che prepara in tempo reale i dati nel formato necessario ad eseguire le predizioni sull'andamento dei successivi 20 minuti. Nella prossima figura vediamo una rappresentazione sintetica di questa architettura, per quanto riguarda la parte dei sensori.

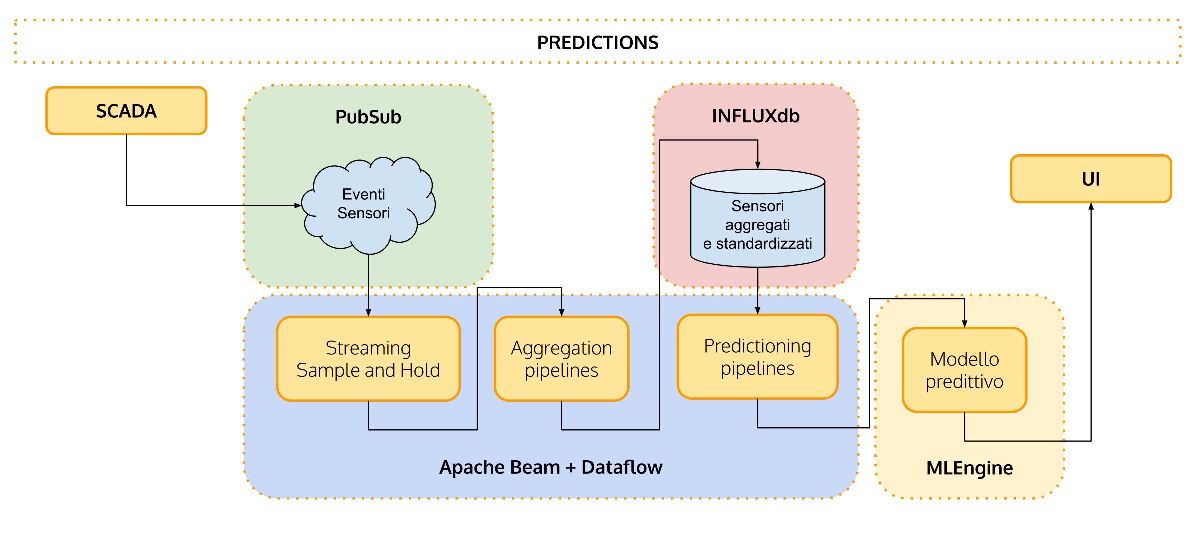


Figura 5.1: Architettura sistema previsionale. Abbiamo gli eventi su PubSub provenienti direttamente da SCADA, questi eventi vengono processati in tempo reale per lanciare previsioni aggiornate con il modello predittivo. La nuvoletta rappresenta un topic PubSub, il cilindro rappresenta un database, i blocchi gialli su Dataflow sono processi di trasformazione dei dati, il blocco SCADA rappresenta il sistema SCADA del cliente, il modello predittivo su MLEngine è il modello allenato, il blocco UI è l'interfaccia utente finale. Ogni elemento dell'architettura realizzata si trova in un piano che indica la tecnologia utilizzata per esso. Su PubSub abbiamo Google PubSub, su InfluxDB abbiamo un database di Influx, su Beam/Dataflow ogni blocco è una pipeline differente, infine MLEngine il blocco è un modello predittivo.

Scopo del progetto realizzato durante il tirocinio è stato in particolare eseguire il cam-

pionamento in tempo reale delle telemetrie dei sensori. L'operazione principale è data dal mantenimento della memoria dell'ultimo valore letto da ogni sensore, per ripeterlo se non dovesse esserci una nuova lettura nell'arco di 15 secondi successivo. Per capire meglio questa problematica vediamo un esempio pratico: consideriamo 3 sensori e un intervallo da 0 a 60 secondi. Dai tre sensori arrivano le seguenti letture:

1. Sensore 1:

    5 secondi, valore 1

    35 secondi, valore 1.5

    50 secondi, valore 2

2. Sensore 2:

    7 secondi, valore 100

    54 secondi, valore 120

3. Sensore 3:

    5 secondi, valore 50

Ogni 15 secondi vogliamo un campione che contiene la lettura più recente di quel sensore.

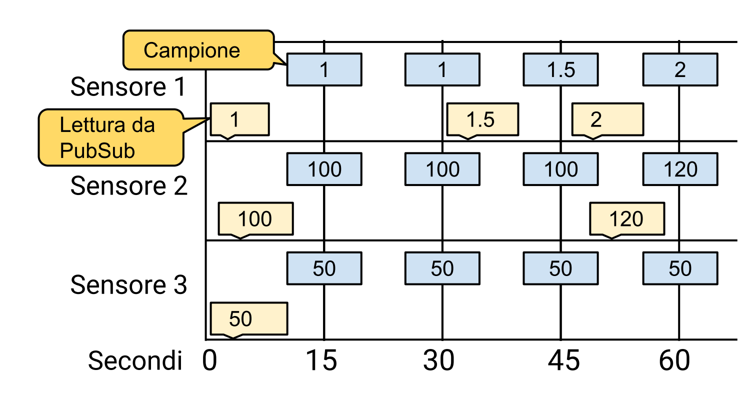


Figura 5.2: Esempio di campionamento Sample and Hold. Sul'asse orizzontale abbiamo il tempo da 0 a 60 secondi. Le nuvolette gialle sono le letture dei sensori in arrivo su PubSub, i blocchi azzurri sono i campioni ottenuti come risultato a intervalli di 15 secondi.

Per ottenere il risultato desiderato, potremmo utilizzare il Windowing per suddividere i dati per sensore, usando il Tag del sensore come chiave, in finestre fisse di 15 secondi. Avremo quindi le letture divise per sensore e per intervallo. Nessun sensore pubblica letture alla stessa frequenza, che è tipicamente superiore ai 15 secondi, sarebbe indispensabile avere una memoria dell'ultimo valore ricevuto. Tutte le operazioni elementari di Beam viste su 3.2 hanno la caratteristica di essere atomiche e senza memoria. Per risolvere questo problema, si è pensato di studiare una nuova caratteristica di Beam, le Stateful ParDo.

5.1 Stateful ParDo

Questo tipo di operazione condivide tutte le caratteristiche della ParDo vista su 3.2.1, in aggiunta si hanno diversi contenitori di stato e timer. Questi sono conservati solamente all'interno della stessa chiave e finestra temporale. Infatti, un vincolo generale di questa trasformazione è l'utilizzo di tuple del tipo $< chiave, valore[, valore_1, \ldots, valore_n] >$.

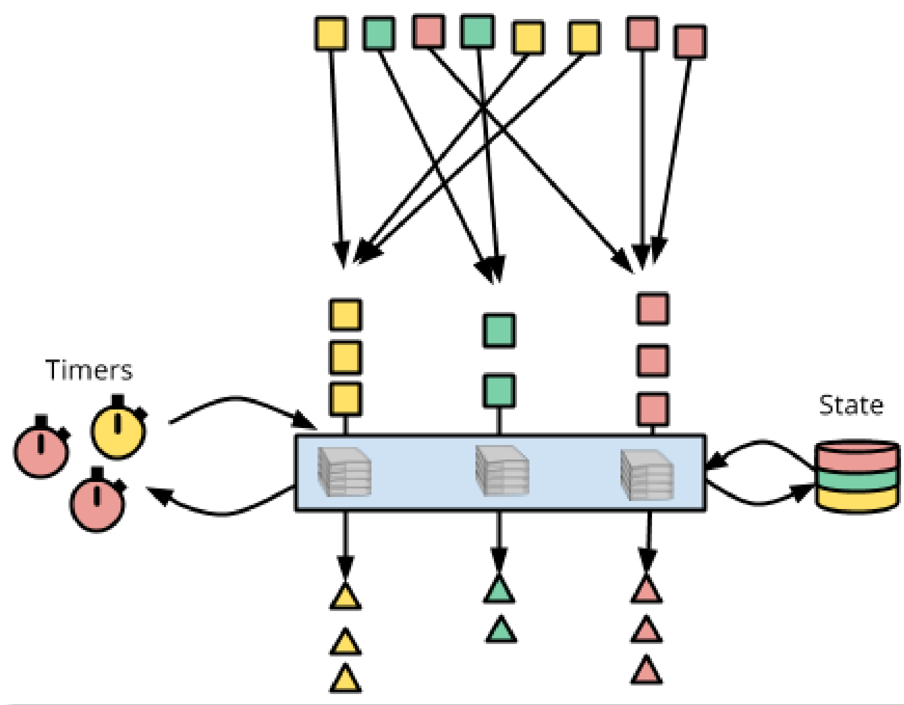


Figura 5.3: Rappresentazione del funzionamento di una ParDo Stateful [27]. Sopra abbiamo i dati in ingresso, con 3 chiavi differenti: "giallo", "verde" e "rosso". Per ogni elemento processato, la Stateful ParDo contiene un proprio stato all'interno della stessa chiave. Ci sono inoltre tre timer: due conservati per la chiave "rosso" e uno per la chiave "giallo". Sotto si hanno gli elementi in uscita dopo il processamento.

5.1.1 Tipi di stato

Sono disponibili due contenitori generici che possono memorizzare lo stato: ValueState e BagState. Il primo memorizza un valore soltanto, mentre il secondo è un buffer non ordinato di elementi.

ValueState

Questo contenitore permette di memorizzare un valore di qualunque tipo serializzabile. Si supponga di voler definire una Stateful ParDo in Java che riceve in input una PCollection come tupla $< chiave, valore >$ di tipo $< String, Integer >$ e in uscita una PCollection di tipo $String$. Inoltre si definisce uno stato di nome *time* che contiene un valore di tipo Instant:

```
ParDo.of(new DoFn<KV<Integer,String>,String>(){
        @StateId("time")
        private final StateSpec<ValueState<Instant>>
                currTimestamp = StateSpecs.value();
        ...
}
```

In questo modo è stata definita la ParDo e il contenitore dello stato. All'interno della ProcessElement di questa ParDo, sarà necessario richiamare lo stato precedentemente definito, come parametro della funzione di processamento:

```
@ProcessElement
public void processElement(ProcessContext c,
@StateId("time") ValueState<Instant> currTimestamp
) {
        .. business logic processamento ..
}
```

Nella business logic si può sovrascrivere il valore memorizzato con `currTimestamp.write(valore)` o leggere l'ultimo che è stato scritto con `currTimestamp.read()`. Nel caso in cui non vi sia alcun valore memorizzato, la *read* restituisce un valore *Null*.

BagState

Questo contenitore permette di memorizzare più elementi. È buona pratica contenere il numero massimo di oggetti memorizzati nel buffer, dato che aumenta l'overhead del processamento. Come per ValueState, può essere definito un contenitore per un tipo qualunque di oggetto serializzabile:

```
ParDo.of(new DoFn<KV<Integer,String>,String>(){
@StateId("buffer")
private final StateSpec<BagState<Event>>
bufferState = StateSpecs.bag();
...
}
```

```
@ProcessElement
public void processElement(ProcessContext c,
@StateId("buffer") BagState<Event> bufferState
) {
.. business logic processamento ..
}
}
```

Nella business logic si possono inserire valori nel buffer attraverso la funzione *add*: `bufferState.add(valore)`, leggere l'intero buffer con `bufferState.read()` e ottenere un oggetto *Iterator* con cui scorrere gli elementi del buffer con `bufferState.read().iterator()`.

5.1.2 Timer

Oltre ai contenitori di stato, vengono offerti due tipi di timer. Questi possono essere impostati ogni qualvolta viene letto un nuovo valore e scattano nel momento in cui si verifica la condizione impostata.

Event Time

L'istante che determina lo scattare di un Event Time Timer è determinata dal Watermark degli eventi ricevuti. Questo significa che una volta impostato il timer a un certo istante, questo scatterà nel momento in cui il Watermark avrà raggiunto tale istante. Nell'esempio che segue si vede un Event Time Timer chiamato `expiry` impostato all'istante di fine finestra dell'evento letto dalla `ProcessElement`.

```
new DoFn<Event, EnrichedEvent>() {
@TimerId("expiry")
 private final TimerSpec expirySpec =
  TimerSpecs.timer(TimeDomain.EVENT_TIME);

@ProcessElement
 public void process(
 ProcessContext context,
```

```
 BoundedWindow window,
 @TimerId("expiry") Timer expiryTimer) {
  expiryTimer.set(window.maxTimestamp().plus(allowedLateness));
  ...
}

@OnTimer("expiry")
 public void onExpiry(
  OnTimerContext context,
  @StateId("buffer") BagState<Event> bufferState) {
  ...
  }
}
```

Il timer `expiry` scatterà qualora il Watermark raggiunga l'istante di fine finestra dell'ultimo Event processato. Ciò potrebbe accadere qualora non venga ricevuto alcun Event per un intervallo superiore a `EVENT_TIME`. Il parametro `BoundedWindow window` permette di conoscere lo stato del Watermark degli elementi processati, nonchè le proprietà sull'eventuale processo di Windowing effettuato. Ogni volta che viene chiamata la `ProcessElement`, il timer viene reimpostato all'istante di fine finestra di quell'Event, con il metodo `timer.set(window.maxTimestamp().plus(RITARDO))`. Quando il timer scatta, viene eseguita la routine dichiarata sulla funzione con il decorator `@OnTimer` .

Processing Time

L'istante che determina lo scadere di un Processing Time Timer è determinato dal tempo di esecuzione della pipeline stessa. Questo tipo di timer permette di controllare il flusso di emissione dei dati, permettendo di impostare un attesa massima prima di avere dei dati in uscita. Nell'esempio che segue si ha un timer `stale` che viene impostato a `MAX_BUFFER_DURATION`, pari a un secondo, ogni volta che viene processato un elemento dalla `ProcessElement`. In questo modo si ha che se trascorre più di un secondo dopo l'ultimo elemento processato, il timer scatta.

```
new DoFn<Event, EnrichedEvent>() {
private static final Duration MAX_BUFFER_DURATION =
```

```
  Duration.standardSeconds(1);

@TimerId("stale")
  private final TimerSpec staleSpec =
    TimerSpecs.timer(TimeDomain.PROCESSING_TIME);

@ProcessElement
public void process(
  ProcessContext context,
  BoundedWindow window,
  @TimerId("stale") Timer staleTimer) {
    staleTimer.offset(MAX_BUFFER_DURATION).setRelative();
    ...
  }

@OnTimer("stale")
  public void onStale(
  OnTimerContext context) {
    ...
  }
}
```

Si noti che la differenza rispetto a un Event Time timer è data dal metodo usato per impostare il timer: in questo caso si usa il metodo `timer.offset(DURATA_TIMER)` `.setRelative();` che imposta il ritardo in termini di tempo relativo rispetto all'istante di processamento. La routine eseguita allo scattare del timer è sempre determinata dalla funzione con il decorator `@OnTimer`

5.1.3 Supporto Python

Dato che tutte le pipeline implementate nell'architettura del progetto visto in questo documento sono scritte in Python, si è pensato inizialmente di realizzare anche questa nello stesso linguaggio. Le Stateful ParDo alla data della realizzazione di questo progetto non erano ancora ufficialmente implementate [28], tuttavia era disponibile una prima

implementazione non documentata[29]. Essa è stata testata ricavando le API dal codice degli Unit Test inclusi alla prima versione. Dopo una prima prova, a runtime si incorre nel seguente avviso:

```
WARNING:root:Key coder FastPrimitivesCoder for transform <ParDo(PTransform)
label=[sample:hold]> with stateful DoFn may not be deterministic.  This may
cause incorrect behavior for complex key types.  Consider adding an input
type hint for this transform.
```

Che suggerisce di utilizzare gli input *type hint* di Python, per evitare un comportamento che "potrebbe non essere deterministico". Infatti, testando con una pipeline che conta gli elementi in ingresso, è emerso che lo stato viene conservato occasionalmente, spesso viene perso. I type hint permettono di eliminare la type inference di Python [30] andando a dichiarare il tipo delle variabili preventivamente. Questo approccio è stato scartato perchè renderebbe l'uso di Python verboso al punto di perdere i vantaggi dati dal linguaggio stesso e potrebbe non aver risolto comunque il comportamento non desiderato, come conferma il fatto che in una versione successiva, disponibile al tempo della redazione del presente documento, ma non ancora al momento della realizzazione del progetto, sono stati corretti dei bug che rendevano le Stateful ParDo in Python non affidabili [31]. Inoltre, sull'implementazione Beam di Python mancano anche i Trigger visti sul capitolo 4, togliendo di fatto il controllo sul flusso di processamento dei dati, affidandolo completamente al Watermark. Si è scelto invece di sfruttare tutte queste caratteristiche e di implementare l'intera pipeline in Java, che le supporta stabilmente.

5.2 Architettura

L'obiettivo che si vuole raggiungere è di avere una pipeline di processamento dati in tempo reale che sia in grado di ricevere in streaming tutte le letture in arrivo dai sensori collegati allo SCADA, filtrare il sottoinsieme interessato e di questo generare campioni a intervalli di 15 secondi. Per verificare la correttezza della trasformazione, si vuole inizialmente scrivere il risultato in tempo reale su un database di Influx.

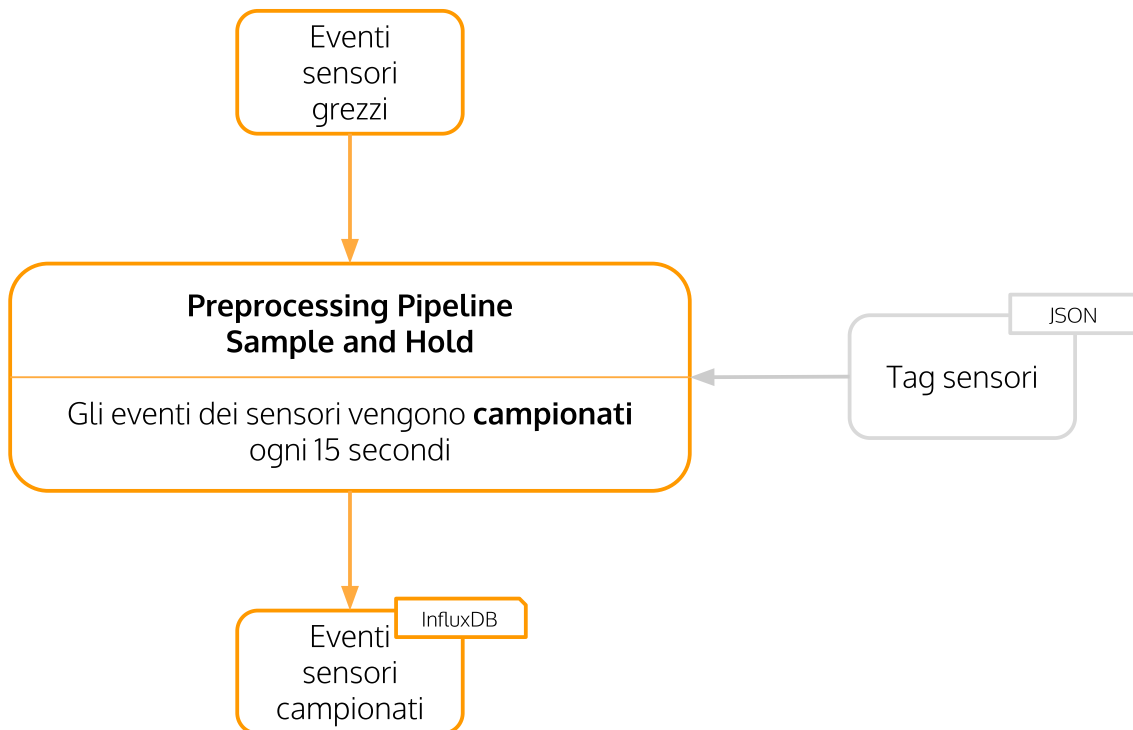


Figura 5.4: Architettura di riferimento: i riquadri in giallo mostrano operazioni eseguite in streaming. Il "JSON Tag sensori" contiene la lista di sensori cui è richiesto il ricampionamento, che rimane fissa per tutta la durata della pipeline.

Per ottenere il requisito richiesto è stata progettata e realizzata una pipeline in streaming con diversi stadi, ogni stadio sarà analizzato passo passo, evidenziando le criticità incontrate. Per manipolare il tipo di dato (eventi di un sensore), è stato creato un oggetto denominato Event che caratterizza la singola lettura. Ogni Event avrà:

- Tag che identifica il sensore
- Valore della lettura
- Data della lettura

Oltre ad Event, che rappresenta il modello del dato, sono state identificate alcune responsabilità principali della pipeline. Queste responsabilità sono state trasformate in classi astratte, classi e funzioni seguendo i principi SOLID [32]:

- Singola responsabilità: ogni oggetto deve avere una sola funzionalità
- Open-closed: aperto alle estensioni, chiuso alle modifiche. Una classe può estendere un'altra, aumentandone le funzionalità ma senza modificarne il funzionamento esistente.
- Liskov substitution, o principio della sostituibilità di Liskov: un sottotipo (un oggetto che ne estende un altro) deve poter essere usato allo stesso modo del tipo di livello superiore
- Interface segregation: Una interfaccia deve contenere collezioni di metodi che vengono usati assieme, senza incorporare metodi non correlati
- Dependency inversion: Le interfacce vengono utilizzate per non far dipendere le classi di alto livello (nel nostro caso, la pipeline) dall'implementazione (per esempio, la ProcessElement di una ParDo).

Si hanno quindi le classi:

- SampleAndHoldPipeline, che contiene la parte principale della pipeline
- PipelineOptions, con tutte le opzioni di esecuzione
- SampleAndHold, con la logica della Stateful ParDo e altre funzioni connesse alla gestione del tempo
- InputOutput, classe astratta con le funzioni *read* e *write* che rappresentano, rispettivamente, il punto di ingresso e di uscita della pipeline
- PubSubInput che implementa il metodo *read* leggendo gli Event da un topic PubSub
- PubSubInflux che implementa il metodo *write* scrivendo gli Event su una tabella di InfluxDB.

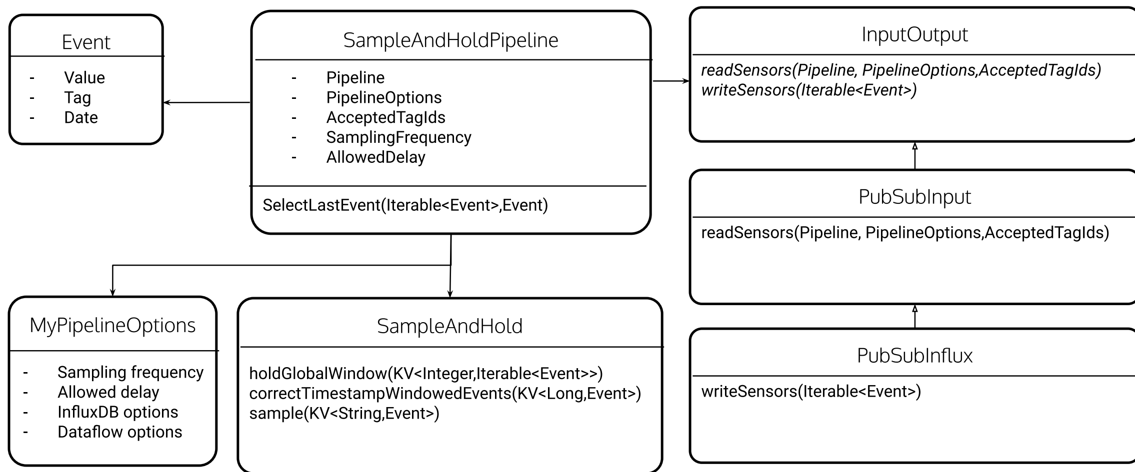


Figura 5.5: UML delle class sintetico. SampleAndHoldPipeline è la classe che contiene la pipeline, Event è l'astrazione della lettura di un sensore, SampleAndHold contiene la logica del campionamento stateful, PubSubInput e PubSubInflux contengono rispettivamente i metodi di lettura da PubSub e scrittura su InfluxDB.

Insieme a queste ci sono anche le classi SensorsUtils, SensorsJsonParser e SampleAndHoldUtils con metodi statici di utilità. La pipeline può essere vista come un flusso unico, che inizia con la lettura da PubSub e poi finirà con il dato campionato correttamente.

5.2.1 Lettura da PubSub e filtraggio

La prima parte si occupa della lettura degli oggetti Event a partire dalle letture in arrivo su un topic PubSub e di eseguire alcune operazioni di filtraggio preliminari.

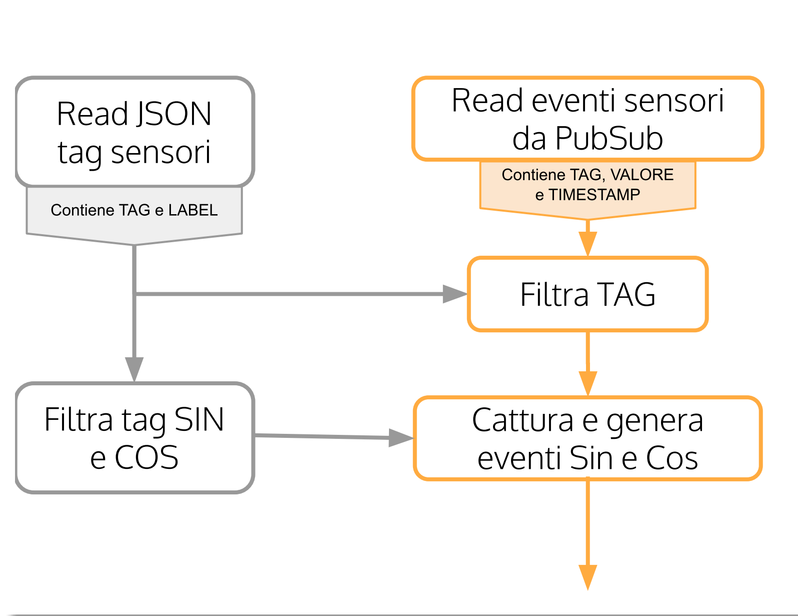


Figura 5.6: Prima parte della pipeline: legge gli eventi da pubsub, filtra i tag interessati dalla lista fornita in JSON, ricava i segnali seno e coseno a partire dai segnali direzione vento.

Viene prima letta la lista dei `TAG` validi, definiti su un file JSON dei sensori da filtrare.

```
Map<String ,String> acceptedTagIDs =
  SensorsJsonParser.parseSensors("sensori.json");
```

Viene poi creata un'istanza di InputOutput. L'implementazione realizzata legge gli eventi da PubSub, riceve la mappa dei TAG accettati e li filtra direttamente. Si ha una PCollection di Event in streaming. Questa PCollection riceverà elementi in tempo reale, mano a mano che arrivano letture dai sensori di interesse.

```
io = new PubSubInflux();
PCollection<Event> allEvents =
  io.readSensors(p, options, acceptedTagIDs);
```

In questo file ci sono anche alcuni `TAG` che finiscono per `_SIN` e `_COS` : sono segnali relativi a sensori di direzione vento, che però non vengono forniti direttamente, è presente invece lo stesso sensore con il valore dell'angolo. Viene prima identificato il sottoinsieme di tali tag, poi calcolato il seno e il coseno di tali angoli, generando i due segnali aggiuntivi.

```
acceptedTagIDs.putAll(
  SensorsUtils.filterSinEvents(acceptedTagIDs));
PCollection<Event> filteredEvents =
  SensorsUtils.filterSinCosEvents(
  allEvents, acceptedTagIDs);
```

Si ha quindi una PCollection di Event che conterrà l'intero insieme dei sensori interessati, compresi quelli fittizi di seno e coseno direzione vento.

5.2.2 Windowing

Il passaggio successivo è l'applicazione del Windowing per la suddivisione degli Event, ovvero delle letture dei sensori, in finestre fisse di 15 secondi.

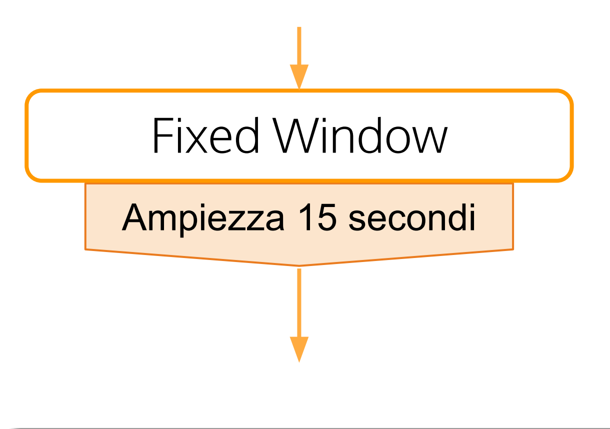


Figura 5.7: Windowing degli eventi sensori. Vengono usate delle Fixed Window di ampiezza 15 secondi.

Duration è una classe di Beam per definire le durate di tempo. Fra le opzioni della pipeline abbiamo la frequenza di campionamento, che è impostata a 15 secondi.

```
Duration samplingFrequency =
  Duration.standardSeconds(options.getSamplingFrequency());
```

Vengono applicate delle Fixed Window di ampiezza 15 secondi:

```
PCollection<Event> windowedEvents =
```

```
filteredEvents.apply(
  "windowing", Window.<Event>
  into(FixedWindows.of(samplingFrequency)
 )
```

5.2.3 Trigger

Nella sezione 4.4 si sono visti i Trigger, che consentono di controllare il flusso dell'elaborazione dei dati. Per il caso d'uso di questa pipeline si hanno due requisiti fondamentali:

- I campioni devono essere in ordine temporale: sarebbe altrimenti necessario riordinarli in seguito, operazione non auspicabile in quanto dispendiosa in termini di computazione e tempo.
- Il ritardo in cui i campioni vengono resi disponibili deve essere ragionevole, altrimenti le previsioni verrebbero fornite con frequenza insufficiente.

Sulla base di queste considerazioni sono stati utilizzati dei Repeated Update Trigger con attivazione Unaligned Delay di 15 secondi, pari alla frequenza di campionamento. Il comportamento è di tipo Discarding, quindi non vengono ripetute letture già inviate al resto della pipeline. Il framework Beam impone di impostare una Allowed Lateness anche per i Repeated Update Trigger (nella sezione 4.4 si era vista per i Completeness Trigger). Essa è stata empiricamente impostata a 5 minuti, tuttavia si può diminuire per ottenere un ritardo inferiore.

```
FixedWindows.of(samplingFrequency))
 .triggering(Repeatedly.forever(
  AfterProcessingTime.pastFirstElementInPane()
   .plusDelayOf(samplingFrequency))) // Unaligned Delay
 .withAllowedLateness(allowedDelay)  // Allowed Lateness
 .discardingFiredPanes()             // Discarding
```

Questa finora è parsa la configurazione con il miglior rapporto prestazioni-stabilità, in sezione 5.4 vedremo altre configurazioni che sono state provate e i risultati ottenuti.

5.2.4 CombineByKey

Il passaggio successivo è l'allineamento del timestamp di ciascun evento all'istante di fine finestra, dal momento che si vogliono avere tutti i campioni a distanza di 15 secondi. Per fare ciò, vengono raggruppati gli Event utilizzando come chiave il loro TAG. Come visto nella sezione 4.1, dopo l'applicazione della GroupByKey avviene l'effettiva suddivisione per finestre temporali, di conseguenza si avranno collezioni di eventi sensori suddivisi per finestra, ma con timestamp variabile all'interno di essa. Si vogliono avere i timestamp degli eventi sensori allineati all'istante di fine della rispettiva finestra temporale. Questo viene realizzato utilizzando la CombineByKey con una funzione di aggregazione personalizzata e una successiva ParDo realizzata allo scopo di allineare il timestamp.

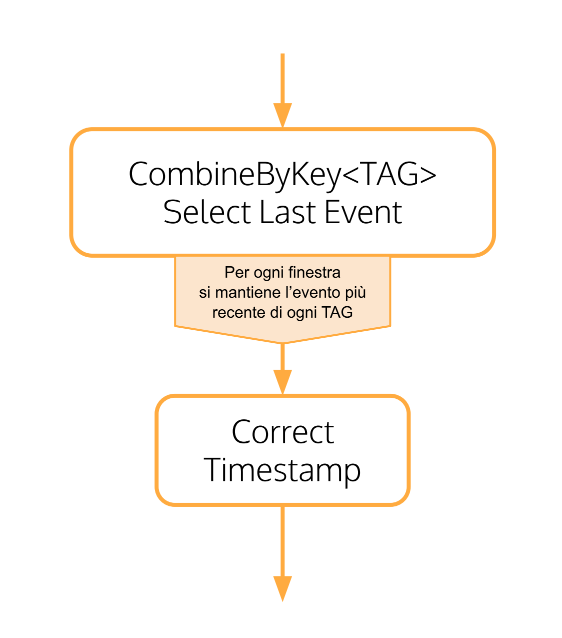


Figura 5.8: Allineamento dei timestamp all'istante di fine finestra. Questo avviene con una CombinePerKey usando come chiave i TAG degli eventi sensori e una successiva ParDo che allinea il timestamp di ciscun evento al suo fine finestra.

La prima operazione in particolare è necessaria dato che una caratteristica di PubSub, come visto su 4.2, è la garanzia di delivery at-least-once, che potrebbe portare ad avere ripetizioni dello stesso evento.

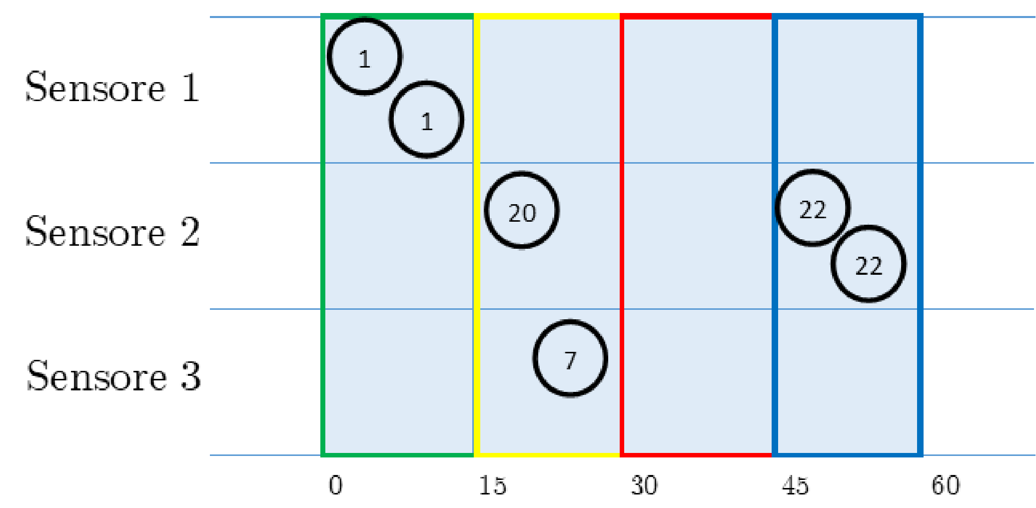


Figura 5.9: Raggruppamento dei sensori per TAG all'interno delle finestre temporali fisse di 15 secondi. Una lettura potrebbe essere ripetuta, ad esempio per Sensore 1 è stata ricevuto due volte il valore "1" nell'intervallo da 0 a 15 secondi, similmente per Sensore 2 con il valore 22 ripetuto nell'intervallo fra 45 e 60 secondi.

Per realizzare questo raggruppamento viene prima estrapolata la chiave da ciascun Event, in modo di realizzare coppie $< TAG, Event >$:

```
PCollection<Event> windowKeyEvents =
windowedEvents.apply("KeyElements",ParDo.of(
 new DoFn<Event,KV<String,Event>>(){
    @ProcessElement
    public void processElement(ProcessContext c) throws Exception {
      String key = c.element().getTag();
      c.output(KV.of(key, c.element()));
    }
 }
));
```

Poi viene utilizzata una CombinePerKey. Si tratta dell'unione di due operazioni:

1. GroupByKey, che in questo caso raggruppa gli Event per TAG e per finestra temporale, ottenendo delle collezioni `Iterable<Event>`

2. ParDo, con il vincolo di avere in ingresso una collezione `Iterable<T>` e in uscita un singolo `T`. Tale funzione di aggregazione combinerà gli elementi nella collezione per ottenere un elemento in uscita. Beam offre alcune funzioni di aggregazione predefinite, come la media. Si possono definire funzioni di aggregazione personalizzate. In questo caso è stata realizzata una funzione che mantiene solo l'Event con timestamp più recente.

La funzione di aggregazione è `SelectLastEvent`, che deve implementare l'interfaccia `SerializableFunction<Iterable<T>,T>`. Questa funzione viene utilizzata per applicare la CombinePerKey. La PCollection `windowEarlyEvents` conterrà quindi per ogni finestra temporale l'evento più recente al suo interno.

```
public static class SelectLastEvent
implements SerializableFunction<Iterable<Event>, Event> {
 // cerca l'evento recente ...
}

PCollection<KV<String,Event>> windowEarlyEvents =
 windowEvents
  .apply(Combine.<String,Event>perKey(new SelectLastEvent()));
```

Dopodichè una successiva ParDo allinea il timestamp degli Event al timestamp di fine finestra. Il metodo statico `sampleEvents` della classe SampleAndHold contiene l'implementazione della ParDo realizzata a questo scopo.

```
PCollection<KV<Long,Event>> sampledEvents =
 SampleAndHold.sampleEvents(windowEarlyEvents)
```

5.2.5 Stateful ParDo

Ciò che si è ottenuto finora sono campioni allineati a distanza di 15 secondi dell'insieme di sensori interessato. Ogni sensore invia un valore a frequenza arbitraria, di conseguenza non avremo un campione ogni 15 secondi di tutti i sensori, ma solo di quelli che hanno inviato una lettura in quell'intervallo.

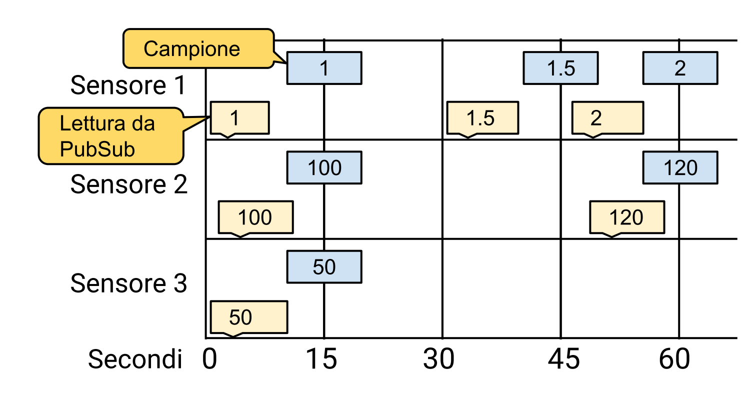


Figura 5.10: Risultato dopo la divisione in finestre e l'allineamento del timestamp. Si ottengono campioni solo per quei sensori che hanno inviato una lettura nell'intervallo di 15 secondi precedente al campione stesso. In questo esempio, sono considerati tre sensori. Nell'intervallo da 0 a 15 secondi, tutti e tre hanno una lettura, quindi si ottiene un campione per ciascuno di essi all'istante 15. Nessun sensore invia una lettura fra 15 e 30 secondi, per cui non si ha nessun campione. Nell'intervallo fra 30 e 45 secondi si ha una lettura solo da Sensore 1, per cui si ha un campione solo per esso all'istante 45, e così via.

L'operazione di mantenere il valore più recente nei campioni successivi costituisce la parte "Hold" del metodo "Sample and Hold". A tale scopo si è pensato di utilizzare una Stateful ParDo che riceve in ordine le letture allineate, tiene memoria dell'ultimo valore letto da ciascun sensore e qualora non ci sia un valore aggiornato emette un campione dell'ultimo valore ricevuto. Vengono quindi innanzitutto raggruppati tutti i campioni dello stesso istante in un'unica collezione.

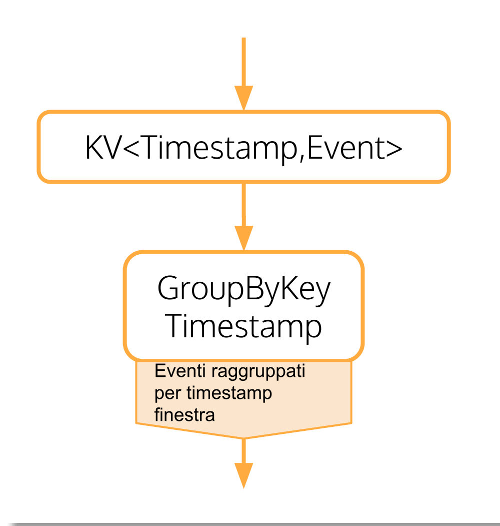


Figura 5.11: Raggruppamento dei campioni allineati con lo stesso timestamp in una unica collezione.

```
PCollection<KV<Long, Iterable<Event>>> groupedSampledEvents =
  sampledEvents.apply(GroupByKey.<Long, Event>create())
```

Come visto nella sezione 5.1 la memoria dello stato è mantenuta solo all'interno della stessa chiave e finestra. Dato che le operazioni da compiere sfruttando il Windowing sono terminate, gli elementi vengono riportati a una unica finestra globale. L'effetto di questa operazione è quello di avere in ingresso alla ParDo gli elementi processati secondo l'ordine e la frequenza determinati dai Trigger che sono stati impostati. Questo richiede un fine tuning preciso di questi, dato che si potrebbero avere elementi fuori ordine (che renderebbe impossibile avere un campionamento corretto) o un ritardo complessivo eccessivo.

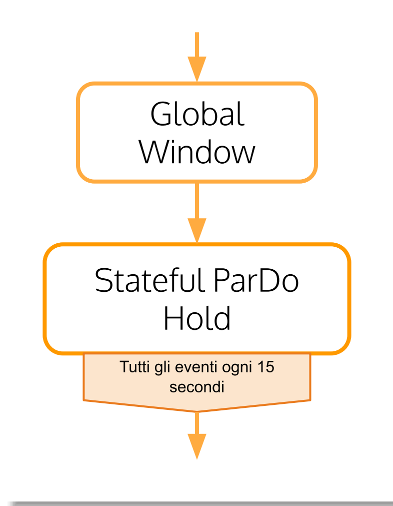


Figura 5.12: Global Window, che riunisce gli elementi in una finestra unica, e la Stateful ParDo vera e propria.

Viene anche applicata una chiave unica, dato che la Stateful ParDo impone in ingresso elementi del tipo `KV<T1,T2>`.

```
PCollection<KV<Integer, Iterable<Event>>> singleKeyEvents =
 groupedSampledEvents
  .apply(Window.<KV<Long, Iterable<Event>>>
   into(new GlobalWindows())) // unica finestra globale
  .apply("singlekey", ParDo.of(
   new DoFn<KV<Long, Iterable<Event>>,KV<Integer, Iterable<Event>>>(){
    // applico chiave singola..
   }))
;
```

A questo punto abbiamo delle collezioni di Event raggruppate per timestamp pronte per essere inviate in streaming alla Stateful ParDo.

```
PCollection<Iterable<Event>> holdEvents =
 SampleAndHold.holdGlobalWindow(singleKeyEvents, acceptedTagIDs);
```

Gli stati mantenuti sono due:

- il timestamp dell'ultima collezione ricevuta e non scartata (perchè in ritardo), chiamato `time`, di tipo ValueState
- il buffer con le letture più recenti di ciascun sensore, chiamato `events`, di tipo BagState.

La ParDo riceve inoltre la lista dell'insieme dei sensori richiesti su `wantedSensorsMap`. Si noti che la `return` non determina il momento in cui saranno mandati in output elementi alla PCollection in uscita, sarà la funzione di output ad aggiungere elementi, attraverso la funzione `output(Event e)` di `ProcessContext`.

```
public static PCollection<Iterable<Event>>
 holdGlobalWindow(
  PCollection<KV<Integer, Iterable<Event>>> timestampEvent,
  final Map<String, String> wantedSensorsMap
  )
   {
    PCollection<Iterable<Event>> sampleAndHoldEvents =
     timestampEvent.apply(
     "holdEvents", ParDo.of(
      new DoFn<KV<Integer, Iterable<Event>>,Iterable<Event>>(){
       @StateId("events")
        private final StateSpec<BagState<Event>> bufferState =
         StateSpecs.bag();
       @StateId("time")
        private final StateSpec<ValueState<Instant>> currTimestamp =
        StateSpecs.value();

       @ProcessElement
        public void processElement(ProcessContext c,
        @StateId("events") BagState<Event> bufferState,
        @StateId("time") ValueState<Instant> currTimestamp
        ) {
         // Business logic ParDo..
        });
```

```
    return sampleAndHoldEvents;
  }
```

Si mantiene una mappa dei sensori attualmente considerati validi su `currentSensorsMap`, dove la chiave è il TAG del sensore. Si memorizza il timestamp della collezione di letture ricevuta su `currEventDate`. Si possono verificare le seguenti casistiche:

1. Si ha una prima lettura, quindi la pipeline è appena stata avviata

   Vengono inizializzati i valori iniziali del timestamp attuale a 15 secondi dopo la lettura corrente.

   Viene riempito il buffer con i valori più recenti. Per i sensori di cui non si ha ancora un valore, lo si inizializza a 0.

2. Si ha una collezione di letture con timestamp pari a quello atteso, ovvero 15 secondi dopo l'ultima ricevuta

   Viene aggiornato il timestamp attuale a 15 secondi dopo la lettura corrente

   Vengono aggiornati i valori nel buffer

   Viene emesso un campione per tutti i sensori

3. Si ha una collezione di letture con timestamp successivo alla precedente, ma con distacco maggiore ai 15 secondi

   Si creano campioni ripetendo i valori memorizzati nel buffer fino a raggiungere il timestamp delle letture aggiornate

   Una volta raggiunto il timestamp pari alla collezione ricevuta, si procede come al punto 1.

4. Si riceve una collezione di letture con timestamp precedente all'ultima

   Sono arrivate in ritardo o fuori ordine: vengono scartate.

Per motivi di spazio non è riportato l'intero codice della business logic, si faccia riferimento allo schema 5.13 e al codice sorgente disponibile esternamente.

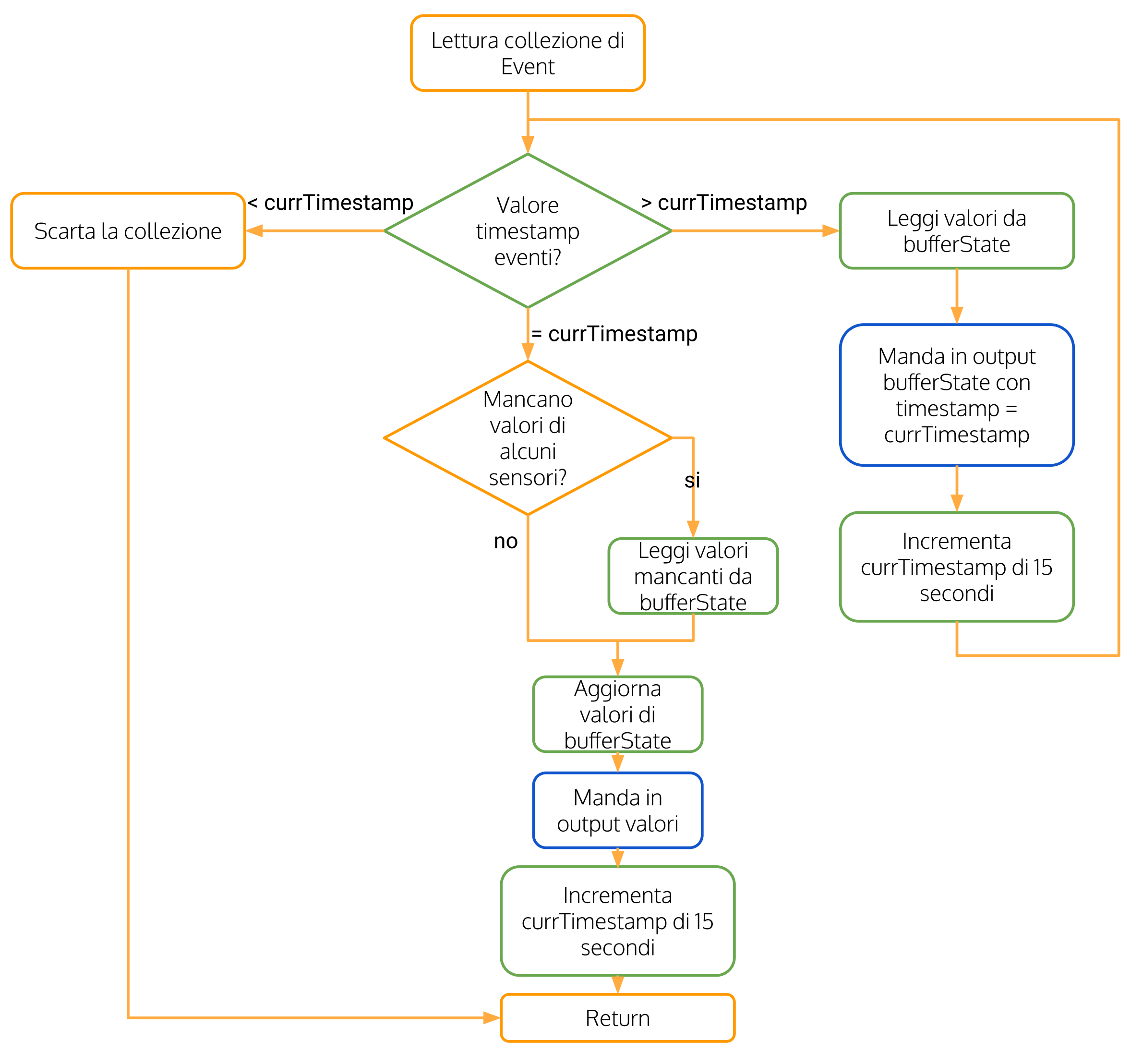


Figura 5.13: Diagramma di flusso della Stateful ParDo, dopo la prima iterazione. I riquadri in verde evidenziano i momenti in cui vengono utilizzati gli stati `bufferState` e `currTimestamp`, quelli in blu evidenziano l'output dei campioni: si noti che la Return non manda alcun valore in uscita. Inizia sempre con la ricezione di una collezione di Event, costituita dall'insieme delle letture in un certo timestamp. Alla prima lettura, vengono inizializzati `currTimestamp` e `bufferState`. Se il flusso dati è gestito correttamente dal Triggering, non dovrebbe verificarsi mai o quasi mai l'ipotesi che scarta la collezione di Event ricevuti.

5.2.6 Scrittura su InfluxDB

Si è pensato di salvare i risultati del processamento della pipeline su InfluxDB, per eseguire dei test a lungo termine in modo di verificarne il corretto funzionamento. A tal fine è necessario realizzare una routine che permetta di salvare gli Event su un database Influx. Dato che al momento non è ancora stata implementata in Beam una trasformazione Input/Output a tale scopo [33] è necessario implementare una ParDo con le primitive di Influx per Java.

```
io.writeSensors(holdEvents, options);
```

Tale ParDo è realizzata nella classe PubSubInflux e utilizza funzioni della libreria `org.influxdb.InfluxDB`.

```
public void writeSensors(
 PCollection<Iterable<Event>> outSensors, MyOptions options) {
  final String influxDBURL = options.getInfluxDBURL();
  final String influxDBuser = options.getInfluxDBuser();
  final String influxDBpassword =
   options.getInfluxDBpassword();
  final String influxDBdb = options.getInfluxDBdb();
  final String influxDBmeasurement =
   options.getInfluxDBmeasurement();
  final Logger LOG = LoggerFactory.getLogger(PubSubInflux.class);
  outSensors.apply(
   // logica scrittura su InfluxDB..
  );
}
```

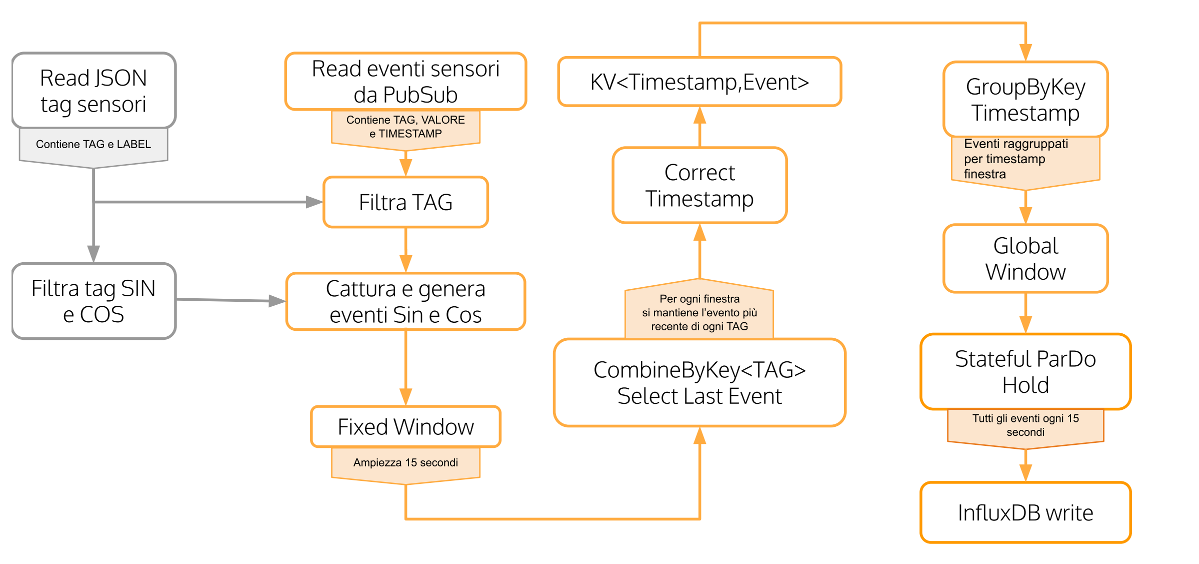


Figura 5.14: Diagramma completo della Streaming Pipeline con la Stateful Pardo realizzata allo scopo di ricampionare i segnali richiesti per la rete previsionale con la tecnica Sample and Hold.

5.3 Simulatore eventi

Al fine di testare il funzionamento della pipeline è stato realizzato un simulatore Python di eventi su PubSub. Questo legge i dati dei sensori grezzi dal database di Influx su cui è stata effettuata l'ingestione, vista nella sezione 2.1, e li pubblica a frequenza periodica sul topic di PubSub dove la pipeline resterà in ascolto. Il timestamp degli eventi sarà mantenuto uguale a quello originale. È possibile determinare la velocità con cui il simulatore pubblicherà gli eventi, se in tempo reale o con velocità superiore. Per poter lanciato necessita dei seguenti parametri:

```
 python simulator/events_simulator.py \
--influx_host $INFLUX_HOST \
--influx_port $INFLUX_PORT \
--influx_user $INFLUX_USER \
--influx_password $INFLUX_PASSWORD \
--influx_read_db $INFLUX_DB_1 \
--influx_read_measurement $INFLUX_M_SEGNALI_1 \
```

```
--start_time $START_TIME \
--end_time $END_TIME \
--project $PROJECT \
--speed_factor $SPEED_FACTOR \
```

In particolare `start_time` determina il timestamp di inizio simulazione, `end_time` quello di fine e `speed_factor` il moltiplicatore di velocità. Uno `speed_factor` di 1 avvierà una simulazione in tempo reale, a 2 andrà al doppio rispetto al tempo reale, e così via.

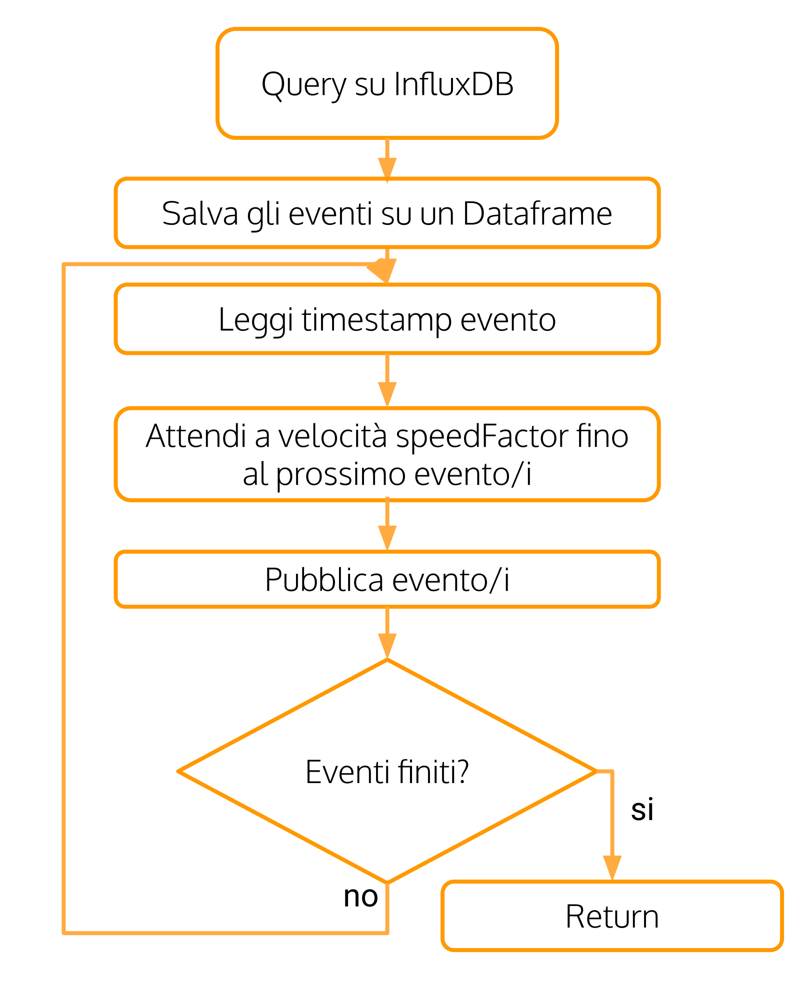


Figura 5.15: Schema logico di funzionamento del simulatore eventi sensori. Il simulatore legge da Influx i dati grezzi dei sensori per pubblicarli periodicamente secondo la velocità impostata.

poi li pubblica dal primo fino all'ultimo secondo la velocità impostata dal moltiplicatore speed_factor.

5.4 Testing e risultati

Prima di ottenere i risultati desiderati si sono effettuate lunghe sessioni di prova e modifica, in particolare per quanto riguarda i trigger, in quanto gli eventi campionati e raggruppati possono essere resi disponibili alla Stateful ParDo fuori ordine, causando quindi la perdita di buona parte dell'informazione.

5.4.1 Prova con Completeness Trigger

Si è provato ad utilizzare un Completeness Trigger, contando sulla previsione euristica del Watermark per l'emissione dei valori nelle finestre. Si ha quindi alla dichiarazione dei trigger dopo il Windowing:

```
FixedWindows.of(samplingFrequency))
.triggering(
  AfterWatermark.pastEndOfWindow()    // Completeness trigger
)                                     //               on-time
.withAllowedLateness(allowedDelay)    // Allowed Lateness
.discardingFiredPanes()               // Discarding
```

Il risultato è che le Fixed Window vengono "chiuse" fuori sequenza, avendo quindi i gruppi di campioni fuori ordine temporale.

```
00:02:29.999Z [[R000561FAND1_1_E_ISF_PMSL002_SFOND01:1.0]]
00:04:14.999Z [[R000538FAND1_1_D_GEN_CANA001_MFLIV01:0.3]]
00:05:14.999Z [[R001136VASD0_1_X_GEN_PLUV001_MFPRC01:0.0], [R001141
00:05:44.999Z [[R000561FAND1_1_E_ISF_PMSL002_SFOND01:0.0]]
00:12:44.999Z [[R000561FAND1_1_E_ISF_PMSL002_SFOND01:1.0]]
00:10:14.999Z [[R001136VASD0_1_X_GEN_PLUV001_MFPRC01:0.0], [R001141
00:08:29.999Z [[R000538FAND1_1_D_ISF_PMSL003_MFVEL01:426.0], [R00053
00:07:29.999Z [[R000538FAND1_1_D_ISF_VASC001_MFLIV02:1.21]]
00:12:59.999Z [[R000547FAND1_1_E_ISF_PMSL001_SFOND01:0.0]]
00:14:59.999Z [[R000547FAND1_1_E_ISF_PMSL001_SFOND01:1.0]]
00:15:14.999Z [[R001136VASD0_1_X_GEN_PLUV001_MFPRC01:0.0], [R001141
00:15:44.999Z [[R000561FAND1_1_E_ISF_PMSL002_SFOND01:0.0]]
00:20:14.999Z [[R001136VASD0_1_X_GEN_PLUV001_MFPRC01:0.0], [R001141
```

Figura 5.16: Log della sequenza di elementi emessa in ingresso alla Stateful ParDo. Il primo valore visualizzato è il timestamp della collezione: l'ordine è in buona parte sparso. Questo causa un comportamento non voluto, rendendo di fatto impossibile il ricampionamento richiesto.

5.4.2 Sliding windows e trigger combinati

Si sono provate soluzioni più complesse. Nella prossima si vede l'uso di Sliding Windows al posto di Fixed Windows, in modo di raccogliere le letture di più sensori, in combinazione a trigger Unaligned Delay di 15 secondi, Completeness on-time e Late Firing a 5 minuti.

```
SlidingWindows.of(SamplingInterval).every(samplingFrequency))
.triggering(
 AfterProcessingTime.pastFirstElementInPane()
.plusDelayOf(samplingFrequency))) // Unaligned Delay
.triggering(
 AfterWatermark.pastEndOfWindow() //Completeness trigger
 .orFinally(                      // late firing
 AfterProcessingTime.pastFirstElementInPane()
 .plusDelayOf(orderingInterval))
)
```

```
.withAllowedLateness(allowedDelay)   // Allowed Lateness
.discardingFiredPanes()              // Discarding
```

Il risultato è simile a quello ottenuto in precedenza, con continui scambi fra i sample, che vengono emessi fuori ordine temporale. Probabilmente la sovrapposizione di diversi tipi di trigger non permette di ottenere il risultato desiderato, dato che non è possibile sapere *a posteriori* quale fra i diversi trigger sia scattato.

5.4.3 Unaligned Delay Trigger

Per ottenere un maggiore controllo sull'ordine, si è semplificata il più possibile la composizione della Window e del trigger, riportando le finestre a fisse e includendo un solo trigger non allineato. Questo trigger, tuttavia, è stato inizialmente impostato a 5 minuti, pari a `orderingInterval`:

```
FixedWindows.of(samplingFrequency))
.triggering(Repeatedly.forever(
AfterProcessingTime.pastFirstElementInPane()
.plusDelayOf(orderingInterval))) // Unaligned Delay
.withAllowedLateness(orderingInterval)   // Allowed Lateness
.discardingFiredPanes()              // Discarding
```

Questo causava nuovamente una consegna fuori ordine temporale delle PCollection di Event. La soluzione finale è data dalla configurazione del trigger Unaligned Delay per scattare dopo 15 secondi rispetto al primo elemento nella finestra, ma tenendo una Allowed Lateness di 5 minuti. In questo modo si ottiene una consegna in ordine della quasi totalità degli eventi. In seguito a un test di 50 ore, si è ottenuto il ricampionamento richiesto dei dati dell'intervallo preso in considerazione.

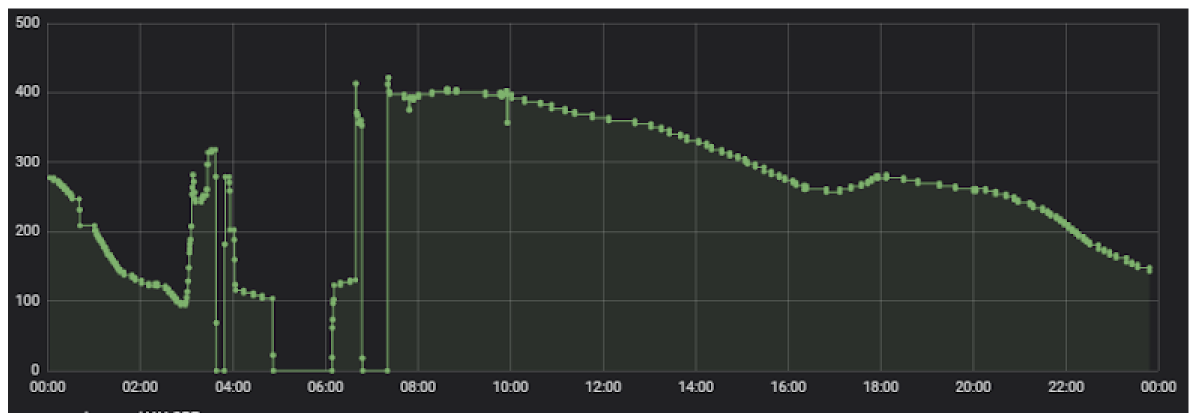


Figura 5.17: Telemetrie grezze di un sensore in una giornata ideale di 24 ore. Si noti che i punti che costituiscono i campioni si trovano a distanze irregolari.

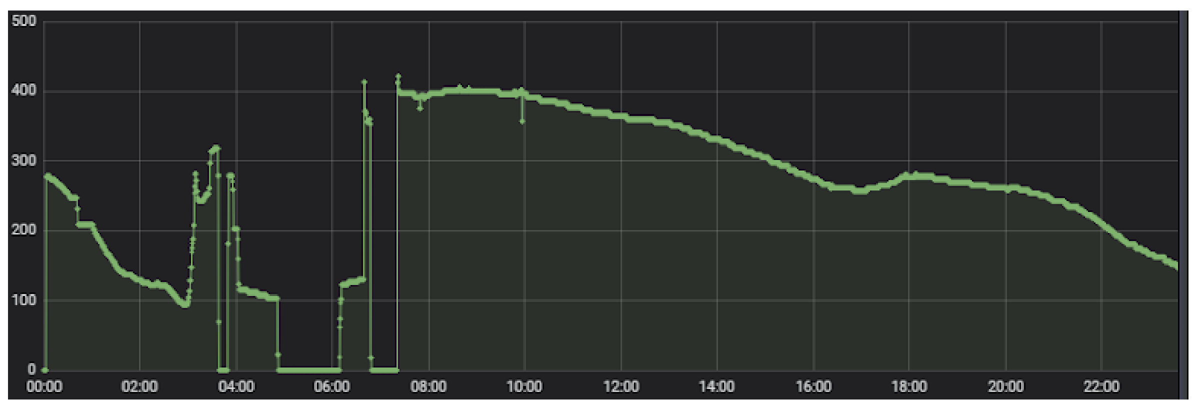


Figura 5.18: Stessa telemetria ma dopo il ricampionamento. Si noti la densità maggiore dei punti che costituiscono i singoli campioni, l'andamento invece è sempre lo stesso, segno di un buon ricampionamento che mantiene l'andamento originale del segnale.

Nel corso di tale test, si è verificato un evento di campioni fuori ordine, che hanno causato un mancato aggiornamento dei livelli di un sottoinsieme di sensori per un minuto circa. Prendiamo in analisi uno di questi eventi.

```
skipping event: [tag=R001134VAND1_1_X_GEN_VENT001_MFDIR01_COS,
skipping event: [tag=R001138VAS41_1_X_GEN_VENT001_MFDIR01_SIN,
skipping event: [tag=R001137VAS50_1_X_GEN_VENT001_MFDIR01_SIN,
skipping event: [tag=R001133VAN10_1_X_GEN_VENT001_MFDIR01_SIN,
```

Figura 5.19: Log della Stateful ParDo che segnala la collezione di eventi fuori ordine.

La simulazione invia segnali dalle 00:00 alle 23:59 del 16/08/2018. Il segnale che è stato ricevuto fuori ordine ha timestamp 14:35:14, mentre l'ultima collezione di letture lette dalla Stateful ParDo era delle 14:35:44. Nelle prossime due figure si vede uno dei segnali coinvolti, che dovrebbe avere un fronte di discesa a -0.846 alle 14:35. Il segnale rimane invece al livello 0.75 fino alle 14:40:15, quando sopraggiunge la successiva lettura.

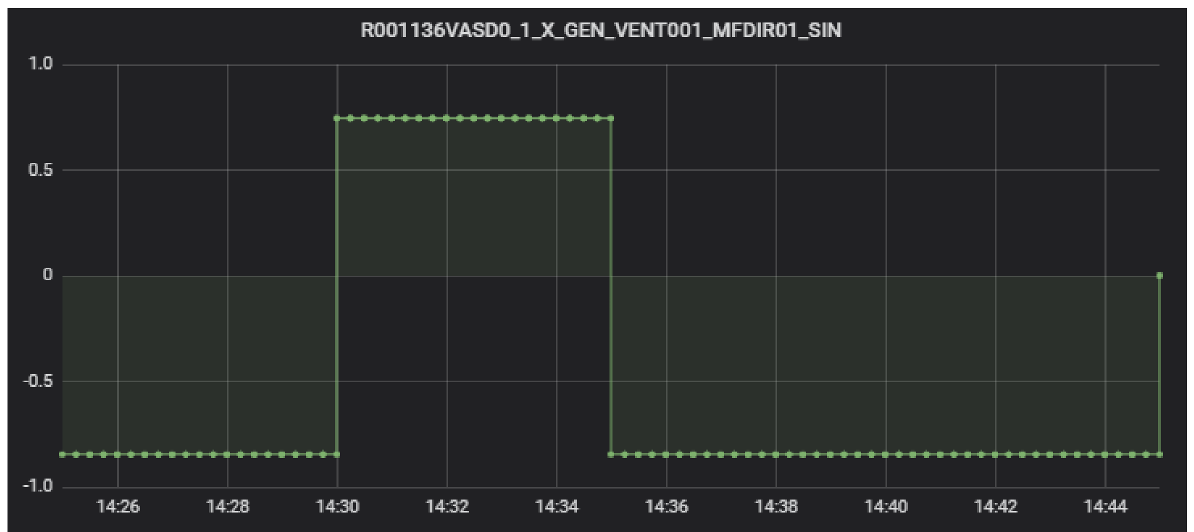


Figura 5.20: Segnale in analisi dopo un ricampionamento corretto.

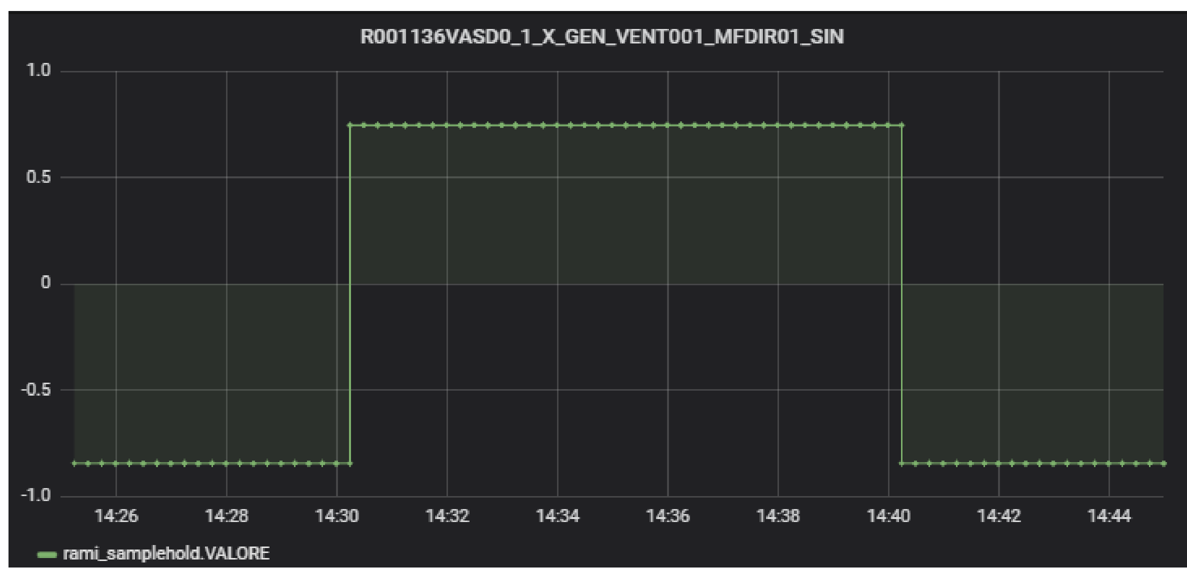


Figura 5.21: Segnale in analisi dopo il caso critico di campione fuori ordine. Il livello del segnale persiste fino alle 14:40:15, in cui vi è una nuova lettura.

Considerato che il sistema è in streaming e in una giornata vi sono 1440 minuti, il livello di affidabilità si mantiene superiore al 99,9%.

5.5 Modifiche e miglioramenti futuri

Questa pipeline è stata studiata e realizzata per il caso d'uso richiesto e svolge correttamente quanto richiesto per esso. Provando ad aumentare lo speed factor del simulatore a 5 volte il tempo reale o più, fenomeni di valori persi a causa del ritardo di processamento, come visto nella parte finale della sezione 5.4.3, accadono più spesso. Questo suggerisce che se le specifiche dovessero cambiare, per esempio richiedendo l'elaborazione di un insieme significativamente più ampio di sensori, mantenendo lo stesso tempo di elaborazione, sarebbe necessario rivedere l'architettura di processamento. Si sono pensate due modifiche che si possono effettuare a questa pipeline in modo di renderla maggiormente scalabile. La prima prevede di dividere la Stateful ParDo in modo che lo stato venga mantenuto separatamente per ciascun sensore. Ciò è possibile trasformando l'ingresso in tipo `KV<String,Event>` dove la stringa è caratterizzata dal TAG della lettura del sensore

di Event. Dato che le Stateful ParDo mantengono lo stato solo fra la stessa chiave e la stessa Window, la separazione dello stato avviene intrinsecamente.

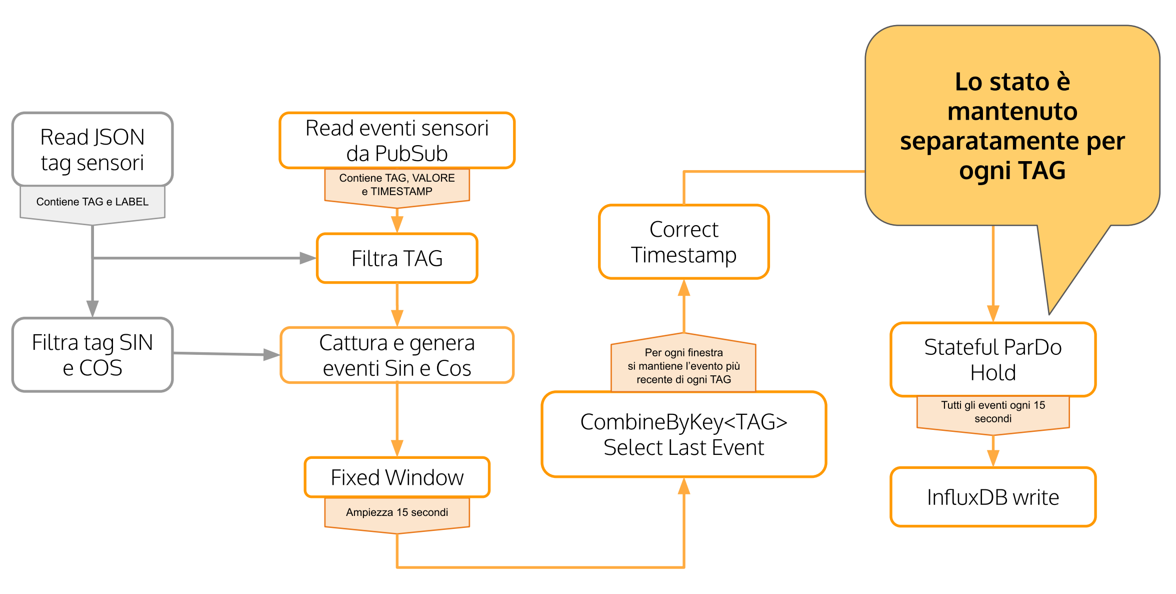


Figura 5.22: Prima modifica proposta per la pipeline in streaming di Sample and Hold. Lo stato viene separato fra i singoli sensori, dato che il TAG è utilizzato come chiave per la Stateful ParDo.

Tale modifica andrebbe a cambiare il comportamento della pipeline, dato che l'uscita non è più uniformata a livello dell'ultimo stadio. Prima infatti si aveva garanzia di ottenere simultaneamente un campione per tutto l'insieme di sensori, applicando la modifica proposta in figura 5.22 si avrebbe un'uscita a intervalli differenti per ogni singolo sensore, qualora venisse mantenuto lo stesso comportamento descritto nel paragrafo 5.2.5. Per ottenere nuovamente un'uscita uniforme si potrebbero utilizzare i Timer delle Stateful ParDo, descritti nella sezione 5.1.2. Con un Processing Time Timer si può imporre il tempo massimo prima che l'output di ogni differente Stateful ParDo abbia luogo, in modo di ottenere una emissione sincronizzata di campioni.

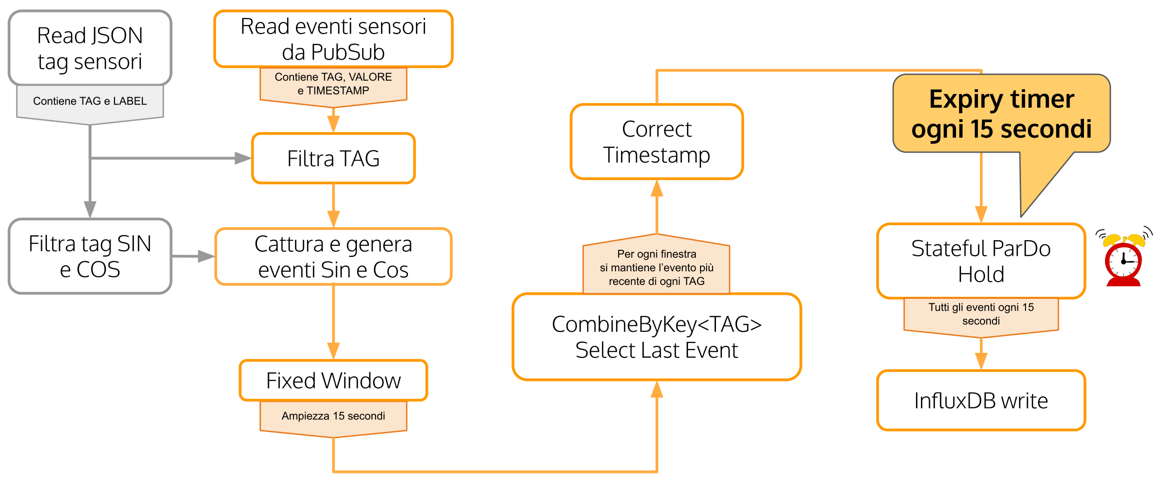


Figura 5.23: Altra modifica proposta. Si introduce un Processing Time Timer che limita il tempo massimo trascorso prima che un output venga prodotto.

Questa pipeline inoltre non gestisce la trasformazione dei dati del meteo, operazione anch'essa necessaria in tempo reale, per cui un lavoro imminente potrebbe essere quello di modificarla aggiungendo un nuovo modello di dato o affiancare la stessa con un'altra simile ma studiata per i dati del meteo.

Conclusioni

Nella realizzazione di un sistema che utilizza tecniche di Machine Learning e, in particolare, Deep Learning, un compito essenziale che non va trascurato è quello della trasformazione dei dati. Infatti, buona parte delle informazioni non possono essere utilizzate nella forma originale in cui vengono prodotte. Si è visto il caso della realizzazione di un sistema che preveda l'andamento futuro su telemetrie di un impianto per la gestione delle acque, sfruttando sia i dati storici sia il meteo. Sono stati sfruttati i dati relativi alle stagioni passate per il training della rete, studiata appositamente per prevedere l'evoluzione futura di queste serie temporali. Prima di ottenere il dataset, è stato necessario eseguire diverse trasformazioni, sfruttando tecnologie allo stato dell'arte di Big Data e Cloud Computing. Questo ha permesso di accorciare drasticamente il tempo di trasformazione e training, rendendo molto più rapida la valutazione della bontà del lavoro svolto per prendere eventuali azioni di miglioramento. Le tecnologie utilizzate si sono rivelate indispensabili anche per rendere disponibili i risultati ottenibili tramite il processo di previsione in tempo utile per l'utilizzatore, che adopererà tali informazioni per intraprendere decisioni critiche. La progettazione e realizzazione di questa architettura in tempi brevi richiede una separazione fra la logica di trasformazione e il processo applicato, distinguendo nettamente fra divisione temporale e semantica dei dati. Tale separazione si è resa possibile utilizzando un sistema realizzato specificatamente per lo scopo, dove le tecniche di gestione temporale delle informazioni sono distinte dalle trasformazioni applicate su di esse. Tutti i metodi e le tecnologie utilizzate, da Tensorflow per implementare la rete neurale a Beam per le trasformazioni sui dati, nonchè le tecnologie di Cloud computing di Google Cloud Platform, sono in rapida evoluzione, insieme alla nascita di nuove applicazioni industriali che richiedono di farne uso.

Appendice A

Tecnologie utilizzate

A.1 Google Cloud Platform

L'intero progetto presentato in questo documento è stato realizzato sfruttando tecnologie della Google Cloud Platform, in modo di sfruttare la potenza computazionale messa a disposizione di Google Cloud. Ogni tecnologia è ottimizzata a uno scopo specifico, qui verranno descritte in maniera estremamente sintetica i principali prodotti utilizzati. Per ciascun progetto si può disporre di un macro raccoglitore che consente di raccogliere tutte le fatturazioni dei consumi effettuati facendo uso dei prodotti per esso.

A.1.1 BigQuery

BigQuery è un data warehouse cloud veloce a elevata scalabilità. È realizzato su una piattaforma serverless, in assenza di un'infrastruttura da gestire. Offre tutte le operazioni CRUD di un database classico, in combinazione a operatori aggiuntivi. Tali operazioni possono eseguite su quantità estremamente grandi in quanto il sistema è ottimizzato per avere alto throughput. È stato introdotto recentemente Bigquery ML che mette a disposizione anche operatori mirati direttamente alla realizzazione di modelli predittivi su dati tabellari.

A.1.2 Google Cloud Storage

Google Cloud Storage (GCS) è una piattaforma di storage in cloud basata sul filesystem di Hadoop. Si può realizzare un *bucket* GCS, che rappresenta l'astrazione di un singolo disco. Il bucket è accessibile dall'URI `gs://nome-bucket/` attraverso le API messe a disposizione. Come il filesystem di Hadoop, è ottimizzato per le operazioni write-once, read-many: sono supportate la scrittura, la lettura e l'eliminazione, ma non lo spostamento diretto di file o cartelle: l'interfaccia utente esegue gli spostamenti tramite una copia seguita dall'eliminazione dalla posizione di partenza. Sono offerti diversi piani tariffari, sulla base dell'allocazione geografica e della frequenza di accesso richieste. Nel caso in cui si volesse effettuare un backup, ad esempio, si può scegliere di realizzare un bucket con un piano a basso costo di stoccaggio ma alto costo di trasferimento.

A.1.3 Compute Engine

Compute Engine rappresenta una piattaforma di virtualizzazione in cloud. Si possono realizzare le singole macchine virtuali, configurare la potenza della singola macchina, capacità di memoria, presenza o meno di scheda video, tipi e dimensione dei dischi, eccetera. Si dispone anche di una pratica interfaccia di collegamento diretto in SSH a ciascuna macchina. In un progetto Google Cloud Platform si hanno dei limiti massimi sul numero di processori a disposizione, dato che Compute Engine è integrato con Dataflow la scalabilità massima è determinata da tale limite. Il costo di una macchina è determinato dalla taglia di processore, di RAM e dalla presenza di schede video (fino a 8) e viene tariffato per tempo di accensione.

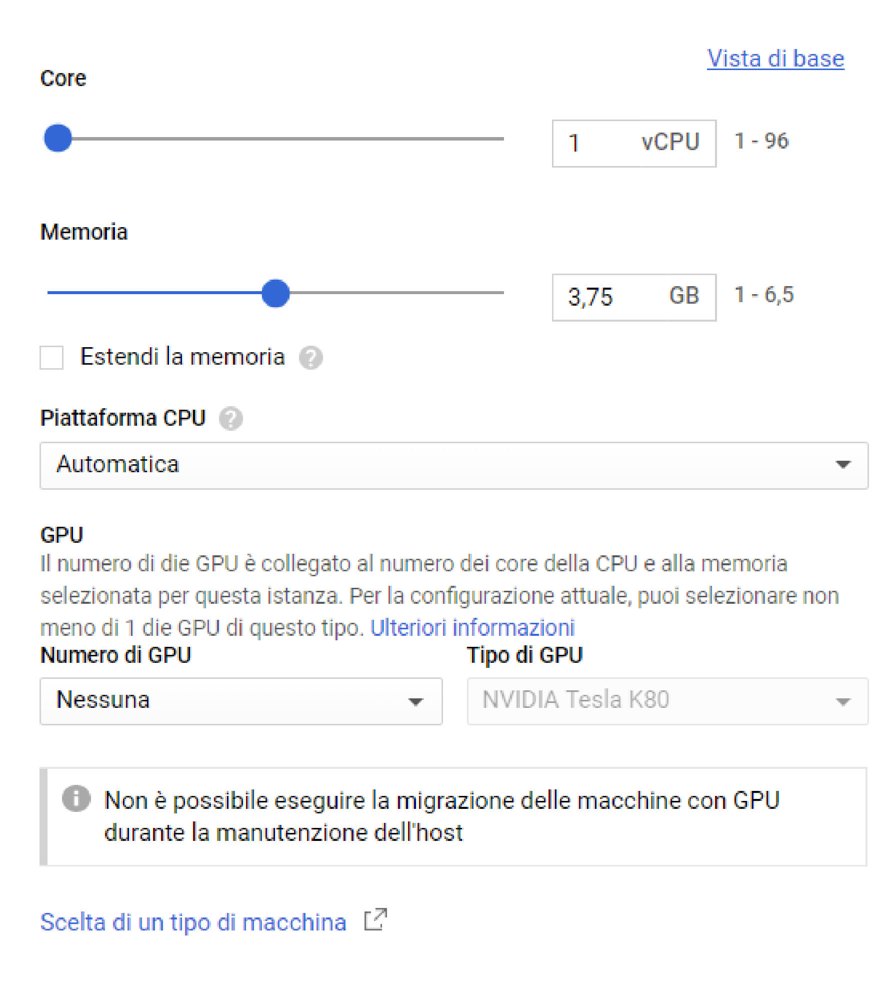


Figura A.1: GCP Compute Engine, schermata di dimensionamento personalizzato macchina. Oltre a una collezione di modelli predefiniti, è possibile scegliere il dimensionamento dei singoli componenti di una macchina. Compute Engine allocherà le risorse richieste sull'hardware a disposizione di Google Cloud.

Una macchina spenta può essere ridimensionata a piacere, per adeguare le risorse al carico richiesto da una certa applicazione, qualora si trovasse in uno stato di carico eccessivo o, viceversa, ridotto. In questo ultimo caso, Compute Engine suggerisce direttamente di ridimensionare le macchine che non utilizzano buona parte del proprio carico, in modo di ridurre sia i costi che gli sprechi, dal momento che una macchina allocata e accesa senza essere utilizzata sta sprecando risorse preziose.

Ridimensiona istanza

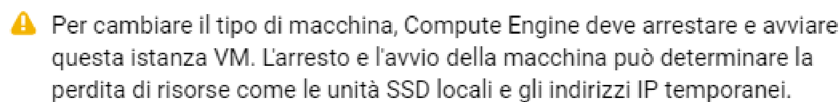
L'utilizzo di CPU da parte dell'istanza recentemente è stato ridotto. Ti conviene passare al tipo di macchina custom (1 vCPU, 7,5 GB di memoria). Ulteriori informazioni

Tipo di macchina attuale

n1-standard-2 (2 vCPU, 7,5 GB di memoria)

Nuovo tipo di macchina

custom (1 vCPU, 7,5 GB di memoria) Consigliato Personalizza



ANNULLA IGNORA SUGGERIMENTO APPLICA

Figura A.2: GCP Compute Engine, schermata di suggerimento ridimensionamento istanza. Per ogni istanza sottoutilizzata, viene suggerito un possibile ridimensionamento per allineare la capacità della macchina al suo carico delle ultime ore.

A.1.4 Google Dataflow

Dataflow permette di lanciare in esecuzione le pipeline di processamento dati facendo uso della capacità computazionale disponibile su Compute Engine. Esso gestirà autonomamente il carico e tutto il processo di creazione e distruzione di macchine virtuali realizzate al solo scopo di scalare dinamicamente la computazione. Per ogni singola pipeline si possono avere metriche accurate sulla quantità di dati processati e sui consumi. Per ogni singolo stadio della pipeline è possibile conoscere il numero di elementi processati, ottenere il log relativo ad essa ed eseguire ricerche avanzate su di esso. In questo

modo si ottiene un alto controllo su flussi complessi con molte fasi di processamento e si possono individuare criticità su possibili casi isolati. Spesso infatti ci potrebbero essere degli *outlier* sui dati non considerati in precedenza, non essendo possibile analizzare ogni singolo record della sorgente che viene processata. Questi outlier potrebbero essere, per esempio, caratterizzati da valori al di fuori di un dominio considerato valido per un particolare campo che viene trasformato.

A.1.5 Piattaforma IA

La Piattaforma IA, precedentemente chiamata MLEngine, è nata per eseguire il training su cloud dei modelli realizzati con Tensorflow e disporre il provisioning dei modelli funzionanti in modo di renderli accessibili da qualunque dispositivo connesso. Dato che anche Tensorflow è un prodotto Google, la Piattaforma IA ruota intorno ad esso e cresce rapidamente allo stesso modo, sono stati resi a disposizione il Hub IA, ricco di materiali condivisi fra la community, integrato con i Notebook. Questi ultimi sono direttamente derivati da Jupyter Notebook, il valore aggiunto è dato dal fatto che il motore Python viene eseguito direttamente su una macchina virtuale in cloud e non è necessario installare alcun server. Inoltre la capacità di computazione delle macchine messe a disposizione sui Notebook di IA Hub è gratuita e non prevede tariffazione. Un altro strumento a disposizione è l'interfaccia di etichettatura (labeling) dei dati.

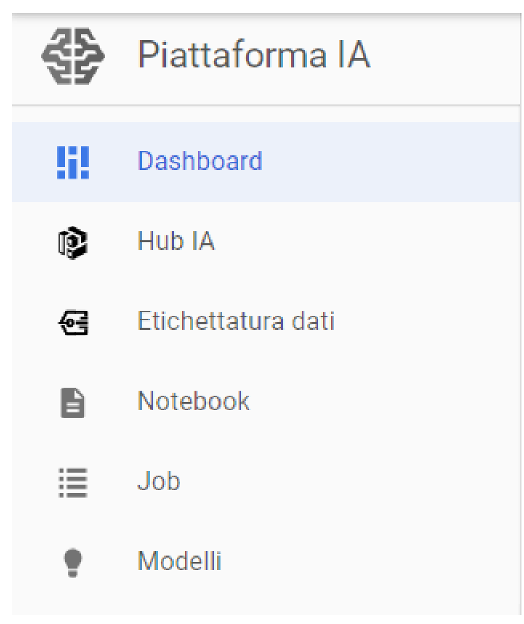


Figura A.3: Piattaforma IA, menu laterale: sono mostrati tutti gli strumenti messi a disposizione. Su Hub IA si può accedere ai notebook condivisi dalla comunità di sviluppatori Tensorflow. Su Etichettatura dati si hanno strumenti utili per il labeling dei dati. In Notebook si possono creare e lanciare con supporto di macchine in cloud i Notebook in Python. Su Job si possono lanciare i training delle reti neurali e visualizzare lo stato e i risultati. Su Modelli si possono caricare i modelli di reti neurali dopo il training per renderli disponibili a qualunque applicativo.

In questo progetto è stata utilizzata principalmente la parte di Job, dato che tutti i training sono stati eseguiti in cloud, in seguito alla preparazione dei dataset.

A.2 Tensorflow e Tensorboard

Tensorflow è un framework realizzato da Google allo scopo di implementare le reti neurali. Le sue API sono in costante cambiamento, in particolare è in corso la transizione da Python 2.7 alla versione 3, dato che entro fine anno verrà deprecata [34]. Mentre nelle prime versioni era necessario implementare manualmente i singoli neuroni con i relativi ingressi rappresentati sotto forma di oggetti tensori, dove un tensore è la singola connessione fra un ingresso, neurone e/o uscita. Ora sono disponibili primitive per la realizzazione di strati già noti in letteratura, ad esempio strati fully connected, con-

voluzionali, ricorrenti, eccetera. Una volta che è stata realizzata la rete ed eseguito il training, è necessario valutare la bontà dei risultati ottenuti, attraverso la valutazione delle metriche raccolte, come la loss (che determina l'entità di variazione del peso nei singoli tensori), il MAE (visto su 3.4), la matrice di confusione, eccetera. Tensorboard è lo strumento principale attualmente utilizzato per avere una valutazione delle metriche ottenute in seguito ai training effettuati su Tensorflow. Ogni singola metrica ottenuta è visualizzabile nel progresso di tutto il training. Sulla base di come sono state aggiunte metriche di valutazione all'architettura della rete si possono avere visualizzazioni più o meno dettagliate.

Bibliografia

[1] C. C. Aggarwal, *Neural networks and deep learning: a textbook*. Springer, 2018.

[2] S. Panzieri. (2015). SCADA. U. R. 3, cur., indirizzo: http://www.dia.uniroma3.it/autom/Reti_e_Sistemi_Automazione/PDF/9NSCADA.pdf.

[3] E. F. I.H Witten, *Data mining: practical machine learning tools and techniques*. Morgan Kaufmann, 2017.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein et al., «Imagenet large scale visual recognition challenge», *International journal of computer vision*, vol. 115, n. 3, pp. 211–252, 2015.

[5] S. Hochreiter e J. Schmidhuber, «Long short-term memory», *Neural computation*, vol. 9, n. 8, pp. 1735–1780, 1997.

[6] C. Olah. (ago. 2015). Understanding LSTM Networks, indirizzo: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[7] I. Sutskever, O. Vinyals e Q. V. Le, «Sequence to sequence learning with neural networks», in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[8] A. Le Guennec, S. Malinowski e R. Tavenard, «Data augmentation for time series classification using convolutional neural networks», in *ECML/PKDD workshop on advanced analytics and learning on temporal data*, 2016.

[9] M. Giuliacci, A. Giuliacci e P. Corazzon, *Manuale di meteorologia*. Alpha Test, 2010.

[10] R. Kimball e M. Ross, *The data warehouse toolkit: the Definitive Guide to Dimensional Modeling*. Wiley, 2013.

[11] A. P. Kefauver e D. Patschke, *Fundamentals of digital audio*. A-R Editions, 2007.

[12] L. Calandrino e M. Chiani, *Quaderni di comunicazioni elettriche*. Pitagora, 2002.

[13] S. M. Ross, *Introduzione alla statistica*. Maggioli, 2014.

[14] *Regola del trapezio*, 2019. indirizzo: https://it.wikipedia.org/wiki/Regola_del_trapezio.

[15] W. W. S. Wei, *Time series analysis univariate and multivariate methods*. Pearson, 1990.

[16] R. Kemker, M. McClure, A. Abitino, T. L. Hayes e C. Kanan, «Measuring catastrophic forgetting in neural networks», in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[17] J. Chung, K. Lee, R. Pedarsani, D. Papailiopoulos e K. Ramchandran, «Ubershuffle: Communication-efficient data shuffling for sgd via coding theory», *Advances in NIPS*, 2017.

[18] *pickle - Python object serialization*. indirizzo: https://docs.python.org/3/library/pickle.html.

[19] T. E. White, *Hadoop: The Definitive Guide*. Yahoo Press, 2012.

[20] *Apache Beam Programming Guide*. indirizzo: https://beam.apache.org/documentation/programming-guide/#creating-a-pipeline.

[21] *numpy.random.RandomState*. indirizzo: https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.RandomState.html.

[22] *Using TFRecords and tf.Example | TensorFlow Core | TensorFlow*. indirizzo: https://www.tensorflow.org/tutorials/load_data/tf_records.

[23] V. Flovik e V. Flovik, *How (not) to use Machine Learning for time series forecasting: Avoiding the pitfalls*, 2018. indirizzo: https://towardsdatascience.com/how-not-to-use-machine-learning-for-time-series-forecasting-avoiding-the-pitfalls-19f9d7adf424.

[24] T. Akidau, S. Chernyak e R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing*. OReilly, 2018.

[25] G. F. Coulouris, *Distributed systems concepts and design*. Addison-Wesley / Pearson Education Ltd., 2012.

[26] T. Akidau, S. Chernyak e R. Lax, *Figures: Streaming Systems Book*, 2018. indirizzo: http://streamingbook.net/fig.

[27] K. Knowles, *Timely (and Stateful) Processing with Apache Beam*, 2017. indirizzo: https://beam.apache.org/blog/2017/08/28/timely-processing.html.

[28] A. Altay, *[BEAM-2687] Python SDK support for Stateful Processing*, 2017. indirizzo: `https://issues.apache.org/jira/browse/BEAM-2687`.

[29] C. Chen, *[BEAM-4594] Implement Beam Python User State and Timer API*, 2019. indirizzo: `https://issues.apache.org/jira/browse/BEAM-4594`.

[30] M. Shannon, *PEP 484 – Type Hints*, 2014. indirizzo: `https://www.python.org/dev/peps/pep-0484/`.

[31] R. Kumar, *[BEAM-7035] Clear() method of OutputTimer is inconsistent*, 2019. indirizzo: `https://issues.apache.org/jira/browse/BEAM-7035`.

[32] R. C. Martin, «Design principles and design patterns», *Object Mentor*, vol. 1, n. 34, p. 597, 2000.

[33] J.-B. Onofré, *[BEAM-2546] Create InfluxDbIO - ASF JIRA*, 2017. indirizzo: `https://issues.apache.org/jira/browse/BEAM-2546`.

[34] *Moving to require Python 3*. indirizzo: `https://python3statement.org/`.

Ringraziamenti

Vorrei ringraziare tutti coloro che hanno reso possibile questo lavoro.
Grazie papà, che hai sempre creduto in me e mi hai sostenuto. Abbiamo vissuto situazioni di grande difficoltà per via della malattia mentale di mamma, che non ne ha nessuna colpa, e non avrebbe mai meritato di fare la fine che gli hanno fatto fare. Sei sempre stato lucido e comprensivo, hai fatto scelte ragionate guardando al futuro e al meglio, pensando a cosa avrebbe portato bene, anche quando chi avevi intorno non era dalla tua parte. Devi essere fiero di tutto.
Ringrazio l'Innovation team di Injenia, una realtà aziendale che investe moltissimo nella ricerca e sviluppo di nuove soluzioni estremamente all'avanguardia e collabora con professori e studenti dell'Università di Bologna per rimanere sempre in contatto con le ultime novità nella ricerca sull'ingegneria dell'informazione. In particolare grazie a Luca Montanari, che mi ha preso subito in simpatia, non solo per i *memini* ma per avermi introdotto al credo in Valliappa Lakshmanan, magnate della Data Science in Google Cloud Platform e in particolare al fantastico mondo del Big Data con le pipeline di Beam e il Cloud Computing. Ringrazio Riccardo, che mi ha seguito con pazienza nelle sfide a me proposte nel progetto in cui sono stato inserito, che ha ascoltato le mie idee e mi ha permesso di realizzarle. Ringrazio Mirko, che mi ha chiarito e illuminato su tutte le caratteristiche del Deep Network da lui realizzato e i principi grazie al quale funziona. Ringrazio Nuccio, per le sue osservazioni costruttive sul fatto che non sempre la soluzione più semplice è la migliore e Gianni, che ascoltando i problemi che stavo avendo con gli Streaming Systems ha tirato fuori l'antologia di Tyler Akidau, senza la quale non avrei mai capito come far funzionare il sistema che stavo realizzando.
Ringrazio tutti i miei amici con cui abbiamo condiviso gioie e dolori nel lungo percorso che porta alla laurea: Andyjobs, il re del trash televisivo, che c'è sempre stato anche al momento del bisogno, insieme a Bea la dea, che nominerei la regina del trash. Livia, che ci vediamo

sempre ma mi manca averti come vicina a due passi da casa, che fra le varie cose mi hai fatto conoscere Nocoldiz in persona. I due Andrei, che mi hanno fatto scoprire la bellezza della loro terra e le delizie che offre. Nicola A. e Riccardo, amicizia storicamente nata grazie all'esame di Ingegneria del Software T, dove abbiamo dato del nostro meglio al punto di creare un legame indelebile. Giuliano, con cui condividiamo la passione per la musica elettronica, siamo stati a uno dei (tragicamente) ultimi concerti dei The Prodigy al completo. Julien, con cui abbiamo condiviso così tanto, viaggiatore impavido e coraggioso, ti auguro sempre il meglio.

Ringrazio anche i miei amici di casa: Piergiacomo, amico da una vita, con cui abbiamo passato di tutto e con cui è nata la passione per i meme ignorantissimi, Elisa C., Martina, Alessia e Nicola S., compagni di scampagnate, avventure e campeggi sempre eccezionali.

Vorrei ringraziare anche molti altri, con cui abbiamo condiviso anche solo un pezzo di questo percorso, in particolare i ragazzi dell'associazione Uni LGBTQ.