ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Realizzazione di un sistema di integrazione dati con tecnologia *Spark*: il caso di Prometeia

Relatore:
Prof. Marco Antonio Boschetti

Candidato:
Simone Romagnoli

Co-relatori
Dott. Luca Nardelli
Dott. Eduart Mustafa

Anno Accademico 2021 – 2022

"Il segreto del successo nella vita è fare della tua vocazione il tuo divertimento." (Mark Twain)

Indice

Elenco delle figure

Abstract

Questa tesi si incentra sul lavoro svolto durante il tirocinio curriculare presso l'azienda *Prometeia S.P.A*, provider di servizi di consulenza, soluzioni tecnologiche e ricerca all'avanguardia [**Prometeia-aboutus**]. Il tirocinio si è svolto presso la sede di Bologna.

Il progetto riguarda l'integrazione di dati (anche detta *data integration*) dei clienti finali di una banca. In particolare, la banca in questione ha richiesto di far convergere tutti i dati dei clienti finali in un unico repository, senza che questi debbano essere prelevati da molteplici sorgenti. In questo modo, è possibile avere dei dati normalizzati, in un formato definito ed in linea con uno standard del prodotto. Ad esempio, uno dei casi d'uso di tali dati coincide con la possibilità di poterli mostrare in un'interfaccia grafica come un *front end* web. Il progetto era già stato realizzato in modo tradizionale mediante l'uso di un database relazionale, ma l'evolversi dei dati (in quantità e disomogeneità) ha richiesto di migliorare le prestazioni del software. Per sopperire a tale problema si è quindi pensato di adoperare una tecnologia in grado di gestire al meglio i *Big Data*: nello specifico, è stata utilizzata la libreria **Spark** di *Apache Hadoop*. La libreria di *Apache Hadoop* consiste in un framework che permette il **processing distribuito** di grandi dataset su un cluster di molteplici computer utilizzando semplici modelli di programmazione [**Apache-hadoop**]. Per sfruttare al meglio tale tecnologia, essa è stata utilizzata in combinazione con il paradigma del **cloud computing** e, in particolare, con i servizi di *Amazon Web Services*.

Capitolo 1

Introduzione al problema

In questo capitolo viene brevemente introdotto il problema per fornire una visione più specifica del lavoro svolto.

1.1 Il problema

1.1.1 Analisi del problema

Il problema in questione è quello dell'**integrazione di dati**, vale a dire, l'unione di informazioni provenienti da diverse sorgenti sotto un'unica vista unificata. Per semplicità e convenzione, ci si riferisce a tale concetto con il termine *data integration*. Tale processo coinvolge una serie di operazioni a partire dalla mera acquisizione del dato, per arrivare alla produzione di un dato pulito e più facilmente fruibile a una qualsiasi entità esterna che vuole accedervi. Le ragioni per cui si effettua *data integration* sono molteplici:

- si vogliono raccogliere dati provenienti da molteplici sorgenti e combinarli poiché semanticamente correlati tra loro;
- si vogliono uniformare dati che assumono formati diversi, ad esempio un campo di una tabella che può assumere un formato di tipo booleano in una sorgente e formato di tipo intero in un'altra, poiché espresso con i valori 0 o 1;
- si vogliono pulire dati considerati sporchi, poiché contenenti informazioni inutilizzate, dati duplicati (quindi ridondanti) oppure campi non valoriz-

zati, cioè incompleti (ad esempio, la valorizzazione di un campo di una tabella con una stringa vuota);

- si vuol fare un *enrichment* dei dati poiché si vogliono ottenere informazioni deducibili da quelle disponibili, ma non direttamente espresse, per via della loro necessità.

Queste, assieme ad una numerosa serie di casi particolari, sono problematiche che al giorno d'oggi influenzano la maggior parte dei sistemi che gestiscono ingenti moli di dati.

Le cause di questi problemi sono da ricercare all'interno della **rivoluzione dei Big Data**. *Big Data* è un termine che fa riferimento a grandi *dataset* dotati di strutture complesse e variegate, che comportano le difficoltà di memorizzazione, analisi e presentazione per ulteriori processi e attività [**big-data-review**]. In particolare, è convenzionalmente noto che i *big data* siano caratterizzati dalle seguenti proprietà:

- **Volume** - vengono prodotti in grandi quantità e dimensione; una sorgente può arrivare a produrre molteplici terabyte al giorno. Si tratta di una quantità di dati che non può essere immagazzinata o elaborata dai sistemi convenzionali e che richiede delle tecnologie *ad hoc*.

- **Velocità** - spesso i dati vengono prodotti ed elaborati da altri processi in tempo reale.

- **Varietà** - i *big data* sono eterogenei; si differenziano per dimensione, formato ed eventuale struttura interna.

- **Veracità** - nel senso che possono essere considerati affidabili, quindi risultano una buona base per analisi aziendali a supporto di decisioni a livello operativo, manageriale ed eventualmente anche direzionale.

- **Valore** - in relazione al fatto che possono essere utilizzati per ottenere un vantaggio strategico. La loro analisi e le previsioni che possono derivarne, possono trasformarsi direttamente in valore per le aziende. Il classico esempio è quello di un'azienda che si basa sull'analisi dei dati relativi ai consumi per prevedere il comportamento di acquisto dei consumatori e

proporre dei prodotti, dei servizi o dei cambiamenti nel business, sulla base di queste previsioni.

L'enorme quantità di dati prodotti a velocità incalzante rappresenta un problema per le aziende *data-driven*, ovvero quelle aziende le cui attività decisionali si basano sui flussi di dati. Nello specifico, per tali aziende è necessario individuare un metodo strutturato e robusto per la gestione dei *big data*, che garantisca anche delle buone prestazioni, in grado di far ottenere un vantaggio competitivo per l'azienda.

1.1.2 Il caso di Prometeia

Prometeia è un'azienda che fornisce servizi di consulenza, soluzioni tecnologiche e ricerca all'avanguardia [**Prometeia-aboutus**]. Le sue sedi sono distribuite sul territorio europeo, nel quale si trovano la maggior parte dei suoi clienti. Il target aziendale consiste in realtà finanziare con mercati estesi anche a livello internazionale. Vista la premessa fatta sui *big data* nella sezione **??**, si può dedurre che quello dell'integrazione di dati sia solo uno dei diversi problemi che *Prometeia* deve affrontare con i suoi clienti. In aggiunta, ogni cliente richiede servizi diversi: per quanto il dominio trattato possa essere simile, ogni cliente ha processi gestiti in maniera del tutto unica. Per fare un esempio concreto, si considerino due banche diverse che possono richiedere un servizio relativo ai rendimenti degli strumenti finanziari per i propri clienti. I clienti della banca possono investire su diversi strumenti finanziari e ottenere, nel tempo, un rendimento relativo alla crescita o perdita degli investimenti. Per quanto gli strumenti finanziari siano un concetto comune a tutte le banche, ciascuna gestisce il portafoglio dei propri clienti in modo diverso, con dati e informazioni uniche rispetto alle altre realtà finanziarie. Questo aspetto costringe *Prometeia* a intraprendere progetti separati per ciascun cliente nonostante trattino problemi comuni. Ovviamente, ogni progetto non viene eseguito disegnando la soluzione dal principio ogni volta, bensì si cerca di seguire una struttura comune per ogni tipologia di servizio richiesto.

Considerando il probleam dell'integrazione di dati, ogni cliente presenta una serie di flussi di dati da integrare, ciascuno con una propria logica e un proprio schema. Questi flussi contengono diverse informazioni, che vanno propriamente

trasformate e arricchite, fino ad ottenere un'unica sorgente che le contiene tutte e che può essere consultata da diversi servizi. Questa tesi approfondisce il caso di un cliente che rimarrà anonimo per motivi di privacy e, in particolare, di un progetto relativo ai rendimenti dei clienti della banca.

Il progetto in considerazione si colloca all'interno dei progetti di ETL di Prometeia. Prometeia considera *data integration* (o ETL) l'insieme di progetti che riguardano l'integrazione di dati tra loro collegati, per ottenere insiemi di informazioni consistenti e connesse tra loro. Il progetto in questione fa parte di un altro macro-progetto chiamato **PFP**, ovvero *Personal Financial Planner*. Il PFP è un prodotto software che le banche utilizzano per aiutare i propri clienti nella gestione di diversi aspetti; in particolare, li guida nella gestione di investimenti, *asset building*, *wealth management*, risparmi e altre decisioni importanti riguardanti le proprie finanze. La parte di ETL realizzata riguarda il calcolo dei rendimenti dei clienti che hanno effettuato degli investimenti. Un rendimento in ambito finanziario rappresenta l'utile o la perdita di un investimento. Per il cliente di una banca è possibile investire in strumenti finanziari, che possono essere obbligazioni, azioni, fondi e tanti altri strumenti di investimento. Ad esempio, il rendimento di un'obbligazione è il tasso di interesse che rende il valore attuale della successione di pagamenti esattamente pari al prezzo attuale [**rendimenti**].

Il calcolo dei rendimenti è un processo che può essere fatto su diverse finestre temporali. Ad esempio, può essere calcolato nell'arco di un giorno o di anni. Le finestre temporali più utilizzate e richieste sono:

- l'ultimo anno (ad esempio, se la richiesta è fatta il 17 marzo 2023, l'ultimo anno va dal 01 gennaio 2022 al 31 dicembre 2022);
- l'ultimo anno fino alla data corrente (corrispondente ai 365 giorni precedenti alla data corrente);
- gli ultimi 3 anni;
- gli ultimi 5 anni.

La richiesta del cliente (la banca) è quella di poter visualizzare tutti i dati relativi ai propri clienti finali su un front end con un'interfaccia intuitiva. I dati vengono forniti direttamente dalla banca in flussi, sotto forma di grandi file

spesso in formato *csv*. Uno dei requisiti del progetto è anche quello di riuscire ad automatizzare il processo di correzione dei flussi sbagliati. Vista la dimensione e l'eterogeneità dei flussi, il caso che il cliente mandi dati non corretti è più che ricorrente. Nei casi normali, quello che si farebbe sarebbe contattare il cliente e richiedere un suo intervento affinchè possa mandare una nuova versione dei dati aggiornata e corretta. Tuttavia, effettuare tale procedura ogni volta che si individua un dato non corretto equivarrebbe ad impegnare tutto il tempo di una o più risorse per gestire le relazione con il cliente.

1.2 Possibili soluzioni

Il processo attraverso il quale viene effettuata l'integrazione di dati, partendo dalla loro estrazione, fino alla loro trasformazione e caricamento finale è detto **ETL**. Il termine *ETL* (*Extract-Transform-Load*) fa riferimento ad insiemi di processi e moduli software responsabili per il reperimento di informazioni da varie sorgenti, la loro pulizia, trasformazione ed integrazione, e il loro caricamento in *data warehouse* aziendali [**etl-def**]. Lo sviluppo dell'*ETL* è potenzialmente uno dei componenti fondamentali per la costruzione di un *data warehouse* sicuro e robusto; infatti, risulta essere una delle attività più complesse e richiede la maggior parte del tempo e risorse aziendali per l'implementazione [**etl-dwh**].

Per un'azienda come Prometeia, che si interfaccia con diversi clienti, esistono due alternative:

1. Realizzare soluzioni *on-premise* per ogni cliente, personalizzando nel dettaglio il processo di integrazione dati ad ogni nuovo progetto.

2. Individuare le porzioni comuni dei modelli trattati dai diversi clienti e realizzare un prodotto vendibile *as a service*.

Non si può dare per scontato che la seconda soluzione sia la migliore in tutti i casi: possono esistere situazioni in cui un'azienda si trovi di fronte a richieste di clienti completamente diverse tra loro e quindi risulterebbe difficile realizzare un prodotto di base o riutilizzabile per soddisfarle. Tuttavia, se le richieste dei clienti fossero accomunabili in parte, si potrebbe creare un insieme di moduli interni da riutilizzare per diversi clienti. L'idea consiste nell'estendere e customizzare un servizio pre-esistente che possa soddisfare requisiti diversi

tra loro, ma riguardanti la stessa problematica o dominio. Applicando tale principio ad un insieme di moduli, si riuscirebbe ad ottenere un miglioramento notevole: innanzitutto, si risparmierebbe lavoro nella realizzazione delle medesime soluzioni riadattate; poi, si otterrebbe uno standard qualitativo che può contraddistinguere l'azienda nel mercato.

Dal punto di vista tecnologico, le diverse soluzioni che si possono individuare vanno valutate in termini di prestazioni e scalabilità. Un sistema di ETL che opera sui *big data* deve garantire:

- che all'aumentare del volume dei dati, non peggiorino esponenzialmente le prestazioni;
- che all'aumento del volume dei dati non corrispondano problemi di disponibilità del servizio, cioè che venga pregiudicato il funzionamento stesso del sistema;
- che il sistema produca dati affidabili in buoni tempi (da valutare a seconda delle dimensioni dei dati);
- che il mantenimento dei dati non garantisca un problema;
- che, in generale, gli *stakeholder* dei dati prodotti possano fare affidamento su di essi.

Quindi, oltre ad organizzare il sistema con un'architettura sufficientemente robusta, è necessario scegliere accuratamente le tecnologie che ne sono a supporto. Ad oggi, diverse tecnologie *open source* sono state sviluppate e messe sul mercato con il diretto scopo di implementare sistemi di ETL. Ma, in generale, la grande distinzione per la realizzazione di sistemi di ETL consiste nella scelta di utilizzo del *cloud*. Infatti, sono diversi i *cloud provider* che sono cresciuti negli ultimi anni fornendo diversi servizi alle aziende, come disposizione di storage o intere piattaforme. Tuttavia, nonostante la scalabilità che quest'ultimi possono offire, alcune realtà possono ancora scegliere di non usufruirne per via di vincoli stringenti come l'utilizzo di determinate tecnologie non compatibili.

1.3 Soluzione adottata

Inizialmente Prometeia gestiva i progetti del PFP separatamente, ovvero che realizzava PFP *ex novo* per ogni nuovo cliente e la manutenzione di questi rimaneva separata. Quindi i diversi moduli di ciascun progetto, come l'ETL o il front end stesso, venivano gestiti separatamente da team diversi. Si trattava di progetti *on premise*, realizzati *ad hoc* per il cliente ed eventualmente installati addirittura direttamente *in loco*. Con il passare del tempo e delle esperienze, è stato notato che la maggior parte dei PFP realizzati presentavano componenti molti simili tra loro. In particolare, si è notata una somiglianza tra i tipi di dati trattati e alcuni algoritmi (ad esempio il calcolo dei rendimenti degli investimenti). Infatti, da un punto di vista macroscopico, ogni banca richiede essenzialmente lo stesso prodotto: dato un insieme di flussi di dati, processare e integrare quest'ultimi per mostrarli nel front end. Quindi, di fatto, le modalità operative dei vari team in Prometeia all'arrivo di un nuovo progetto corrispondevano a prendere un altro PFP somigliante e fare modifiche per adattarlo al nuovo. Questo approccio, in realtà, presenta una serie di problemi:

- Tale approccio duplica l'*effort* impiegato per la realizzazione dei progetti. Il riadattamento di un modulo software di fatto significa l'applicazione di modifiche che possono apportare nuovi problemi, richiedendo ulteriore tempo da parte dei vari membri dei team.

- Provoca una duplicazione del codice non indifferente. Duplicando il codice, si vanno spesso a creare dei rischi difficili da mitigare e si crea un prodotto dalla difficile manutenzione, sia dal punto di vista della complesità che dell'*effort* richiesto. Ad esempio, anche per fare *bug fixing* di una dipendenza o di una feature, bisognerebbe ricordarsi di applicare la modifica in ogni parte duplicata.

- Ci si potrebbe rendere conto troppo tardi che il progetto di partenza scelto non sia quello più adatto.

- In caso di mancata documentazione sarebbe necessario reperire i responsabili del progetto nel caso sia necessario avere supporto.

Quindi in Prometeia è nata l'esigenza di cambiare approccio al problema. Nello specifico, è stato fatto partire un progetto interno chiamato **Baseline**.

Il progetto Baseline corrisponde a una serie di progetti che sono condotti internamente da Prometeia e che quindi vengono considerati come **prodotto**. Per "prodotto" si intende una particolare funzionalità che Prometeia fornisce ai clienti come pacchetto base che può essere esteso a seconda delle esigenze del singolo cliente. Alcuni esempi di moduli software realizzati per Baseline sono:

- Baseline Batch - un modulo che implementa un Enterprise ETL prendendo come riferimento un insieme di flussi base e implementato utilizzando la tecnologia Spark.
- Baseline PFP - un progetto che presenta sia una parte di back end, sia una parte di front end, realizzate rispettivamente con tecnologie *Spring* e *Angular*. Questo progetto, di fatto, rappresenta il nuovo punto di partenza dei nuovi PFP ed è quindi estendibile e customizzabile a seconda delle richieste.

In sostanza, Baseline è una *suite* di soluzioni software che il cliente può comprare e decidere di estendere a seconda delle proprie esigenze.

Oltre a Baseline, Prometeia ha anche deciso di spostarsi sul *Software as a Service*. Vale a dire, che Prometeia ha individuato una serie di moduli software vendibili come *standalone* poiché integrabili con diversi applicativi dei clienti. Quindi, a differenza dei vari pacchetti che richiedono comunque un'integrazione, la parte *SaaS*, chiamata **DWM** (*Digital Wealth Management*), non è altro che un insieme di servizi REST che Prometeia è in grado di vendere nella loro interezza e che le banche possono eventualmente integrare nei loro applicativi non direttamente gestiti da Prometeia. Tra questi si possono trovare:

- Il servizio Catalogo Strumenti, che serve a catalogare tutti gli strumenti finanziari, che quindi risultano integrati e reperibili *as a Service*.
- Il servizio Rendimenti, che permette di ottenere il calcolo dei rendimenti su diverse finestre temporali, come discusso nella sezione **??**.
- Il servizio Customer, che presenta un'anagrafica completa dei clienti della banca.

Ognuno di questi servizi espone varie informazioni che sono utili al PFP, ma anche a eventuali esigenze nuove o esterne che la banca può avere e che non

vengono gestite da Prometeia. Un possibile caso d'uso può essere: una banca ha un suo applicativo *private* che ha bisogno del catalogo degli strumenti completo e integrato. Prometeia può quindi vendere solo il catalogo strumenti come servizio REST in un modulo *as a Service*. In sostanza, la differenza tra Baseline e DWM consiste nel fatto che il primo è una libreria **estendibile**, creata appositamente per soddisfare requisiti comuni a più progetti; mentre i moduli DWM sono dei servizi **a sè stanti** sfruttabili *as a Service* sia da clienti, sia da Prometeia stessa all'interno di progetti.

Da quando esiste Baseline, i nuovi PFP hanno sempre fatto uso di questi moduli *SaaS*. Per il progetto di tirocinio in considerazione, invece, si tratta di un caso leggermente diverso. Infatti, Baseline è nato dopo la realizzazione del PFP per il cliente in questione, che è attivo e operativo da diverso tempo. Quindi il PFP in questione non è basato su Baseline, nè fa uso di moduli *SaaS*. Nell'ultimo anno però, i nuovi sviluppi per tale cliente hanno integrato una parte del mondo SaaS di Prometeia. Un esempio è il *Profiling*, ovvero il software di profilazione *multi-tenant* della clientela. *Profiling* è un progetto back end, venduto come *SaaS*, che espone servizi per creare profili, monitorarli per la gestione del rischio, e tante altre funzionalità. A questo viene poi associato un front end customizzato. Oltre a *Profiling*, è recentemente partita l'integrazione del modulo Baseline dei rendimenti.

Per riassumere, al momento dell'inizio del tirocinio, erano già state implementate le seguenti componenti:

- Un modulo che implementa l'Enterprise ETL, che include una parte integrata dal prodotto di Baseline più le varie integrazioni a progetto. Questa ha il delicato compito di trasformare i dati della banca in modelli utilizzabili a livello di servizio e delle altre catene di batch.

- Il catalogo strumenti esposto sia come batch interno al progetto, sia come servizio invocabile.

- Il calcolo dei rendimenti, esposto sia a livello di servizio, sia come parte batch a prodotto customizzato per adattarlo alle richieste.

All'interno di questo contesto è iniziata l'attività di tirocinio. Vista la durata estesa del rapporto con Prometeia, l'attività di tirocinio non si è limitata alla

realizzazione di un solo obiettivo. Infatti, oltre ad apprendere ed esplorare la tecnologia Spark, il tirocinio ha mirato all'espansione di progetti già iniziati e al lancio di nuove parti. Nello specifico, le attività intraprese e discusse nei capitoli seguenti riguardano:

- La continuazione della realizzazione di ETL per un cliente con tecnologia Spark, assieme al calcolo dei rendimenti. Tale progetto era già iniziato al momento dell'inizio del tirocinio, ma necessitava di supporto. Quindi tale attività è consistita nel ricevimento quotidiano di task dagli analisti che interagivano direttamente col cliente. Per portare a termine questi task, è stato necessario aggiungere feature a progetto, ma anche effettuare modifiche a prodotto per adattarsi alle richieste.

- L'inizio di una rivisitazione di un progetto per il medesimo cliente, avvicinandolo al mondo del prodotto con tecnologia Spark. In particolare, la realizzazione di alcune parti di un PFP con la prospettiva di abbattere i lunghi tempi d'esecuzione che possono essere ridotti. Questa parte rimarrebbe comunque un progetto realizzato *ad hoc* in quanto non segue la stessa logica del PFP di Baseline. Comunque l'obiettivo rimane il miglioramento dei tempi grazie all'utilizzo di nuove tecnologie e sfruttando i nuovi servizi DWM.

- L'inizio di un progetto per Baseline, riguardante la realizzazione di un batch che implementi il comportamento di un ETL tipico dei progetti per i clienti di Prometeia. L'obiettivo finale di questa parte consiste nella realizzazione dei vari step, quali l'ETL di flussi di dati, il catalogo degli strumenti finanziari, il calcolo dei rendimenti, e altre parti. Nello specifico, il tirocinio ha coperto per intero la realizzazione dell'ETL e del catalogo degli strumenti.

Dal punto di vista tecnologico, *Prometeia* fa generalmente uso di tecnologie *open source*: uno dei pregi dell'*open source* è quello dell'affidabilità che deriva da un continuo miglioramento delle applicazioni. Mentre i software proprietari legano la fedeltà del cliente attraverso le licenze, l'*open source* mira alla soddisfazione dell'utilizzatore. I fornitori di soluzioni *open source*, infatti sono soliti fornire servizi, consulenza e formazione. La qualità è garantita dalle

relazioni che si instaurano tra gli utilizzatori, i proprietari e la community di programmatori grazie a corsi, documentazioni aggiornate, ma anche dalle continue correzioni degli errori [**open-source**]. In particolare, per la realizzazione dei sistemi sopra citati, è stato deciso di fare affidamento al *cloud* e ai servizi di *Amazon Web Services*. Tale scelta non è nuova a Prometeia: infatti, è stata l'ennesima conferma di un paradigma aziendale già utilizzato che ha portato ottimi risultati. La combinazione del *cloud* con la possibilità di estendere librerie interne si è rivelato estremamente efficiente da diversi punti di vista: in questo modo non solo è possibile effettuare lo *start-up* dei sistemi in maniera veloce, ma risulta anche comodo per allocare nuove risorse su nuovi progetti; la modularità dei sistemi realizzati rende molto semplice anche la formazione di personale che deve iniziare a lavorare su progetti appena iniziati.

Capitolo 2

Analisi tecnica

In questo capitolo si effettua una analisi tecnica, relativa alle tecnologie adoperate, per ottenere un'ampia panoramica sul loro utilizzo e sulle loro funzionalità.

2.1 Il framework: Spark

La tecnologia fondamentale che costituisce il fulcro del progetto è Spark. I programmi Spark richiedono l'impiego di 5 entità:

- un driver program;
- un cluster manager;
- molteplici worker;
- degli executor;
- dei task.

Un *driver program* è una applicazione che usa Spark come libreria e costituisce il punto d'ingresso di tutto il sistema; volendo, costituisce il *main* del programma. Per connettersi al *cluster manager*, e quindi agli eventuali cluster, è necessario inizializzare uno `SparkContext` che stabilisce la connessione. Il *cluster manager* inizializza dei nodi *worker* che forniscono memoria e storage all'applicazione Spark. Su ogni *worker*, vengono create delle JVM (*Java Virtual*

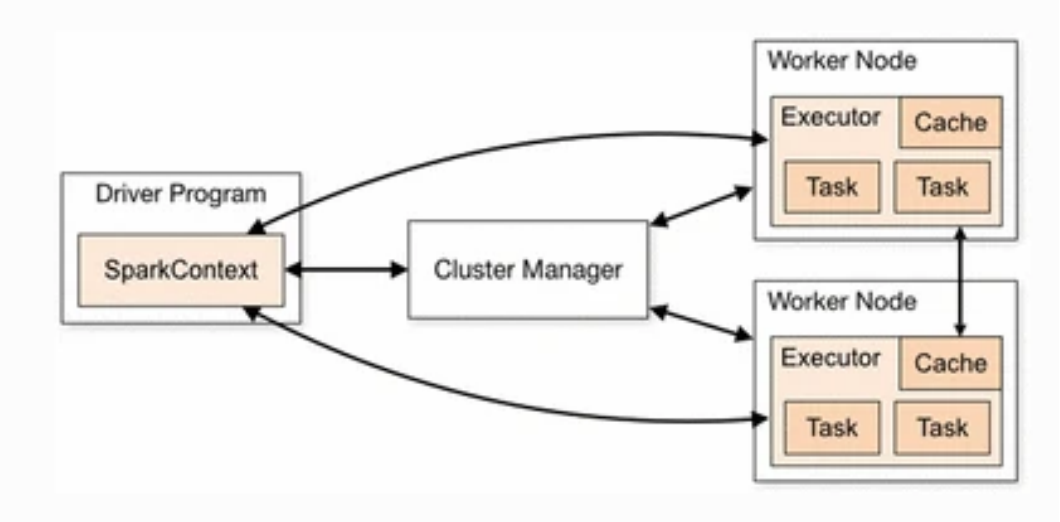

Figura 2.1: Funzionamento di un programma Spark.

Machine) che costituiscono gli *executor*. Ogni *executor* è in grado di eseguire molteplici *task* assegnati dal *cluster manager*. Un programma Spark è in grado di lanciare molteplici **job**, dove un *job* rappresenta un insieme di computazioni necessarie per restituire un risultato al *driver program*. Un *job* può essere suddiviso in **stage**: nello specifico, Spark crea un *directed acyclic graph* (DAG) di *stage*, e suddivide ogni *stage* in molteplici **task [big-data-spark]**. I task possono essere visti come la più piccola unità di lavoro eseguibile dagli executor. Una volta ottenuto il risultato dei task, gli *executor* possono inoltrarlo al *driver program*. La figura **??** è una rappresentazione della comunicazione tra queste entità.

2.1.1 Storia

Il progetto Spark è nato nel 2009 alla *Berkeley's AMPLab, University of California* per poi essere donato alla *Apache Software Foundation* nel 2013 **[big-data-spark]**. Il progetto consiste di diverse componenti principali, tra cui *Spark SQL*, *Spark MLlib* per il machine learning, *GraphX* e *Spark Streaming*. Storicamente, Spark ha introdotto un'astrazione per modellare algoritmi e processi con migliori prestazioni: i **Resilient Distributed Dataset** (*RDD*). Un RDD è una collezione di record partizionati e *read-only*; rappresenta una struttura dati parallela e *fault-tolerant*,, il cui partizionamento e contenuto può

essere manipolato attraverso una serie di operazioni [**big-data-spark**]. Tutte le componenti di Spark sfruttano la astrazione degli RDD; tuttavia, il progetto Spark è in continua evoluzione. Ciò ha portato a diverse feature che hanno sostituito gli RDD: tra queste, vi è l'introduzione dei **DataFrame**, facenti parte del modulo *Spark SQL*. Un *DataFrame* è concettualmente equivalente ad una tabella di un database relazionale, ma distribuita e con trasformazioni ottimizzate. La feature più importante rispetto agli RDD è quella dei dati organizzati in colonne.

Un altro miglioramento è stato introdotto nella versione Spark 1.6 con i *Dataset*. In particolare, l'obiettivo dei *Dataset* è quello di unire i vantaggi degli RDD e dell'ottimizzazione dell'*execution engine* di *Spark SQL*.

2.1.2 Datasets

Spark permette di eseguire operazioni parallele su molteplici cluster remoti. Il vantaggio che questa libreria porta è quello di scrivere codice parallelo con una sintassi sequenziale, effettuando operazioni tipizzate su Dataset distribuiti. Un *Dataset* è una `collection` fortemente tipizzata di oggetti che possono essere trasformati in parallelo e sono mappati ad uno schema relazionale. Ogni *Dataset* ha anche una vista non tipizzata chiamata **DataFrame**, che consiste in un `Dataset` di Row; una `Row` rappresenta una riga di output di un operatore relazionale.

I dati sono distribuiti sui cluster, e *Spark* rende completamente trasparente il partizionamento agli sviluppatori. In particolare, questo paradigma distingue le operazioni eseguibili sui *Dataset* in **Azioni** e **Trasformazioni**. Le trasformazioni sono operazioni che producono nuovi *Dataset* come output; tra le trasformazioni, alcune delle più comuni sono:

- *map* - applica una trasformazione a tutti gli elementi del *Dataset*.
- *filter* - permette l'applicazione di un filtro sugli elementi del *Dataset*, rimuovendo quelli che non rispettano il predicato passato come argomento.
- *select* - esegue una espressione-colonna sul *Dataset*, ritornandone uno con le colonne specificate nell'espressione.
- *union* - unisce gli elementi di due *Dataset*.

- *join* - esegue l'operazione di *join* tra due *Dataset*. Tale trasformazione permette di specificare una *join expression*, che determina i campi da utilizzare per l'operazione. Inoltre, permette di definire il tipo di *join* della trasformazione; i tipi di *join* sono elencati e descritti nella sezione **??**.

- *withColumn* - aggiunge una colonna al *Dataset* con una *column-expression*.

- *drop* - rimuove dal *Dataset* la colonna specificata.

Invece, le azioni sono operazioni che provocano delle computazioni le quali ritornano dei risultati. Nello specifico, richiedono di reperire in locale il contenuto distribuito dei *Dataset*. Alcune azioni sono:

- *collect* - ritorna un array con il contenuto del *Dataset*.

- *count* - ritorna il numero di elementi del *Dataset*.

- *foreach* - esegue una funzione su ogni elemento del *Dataset*.

- *reduce* - riduce il *Dataset* applicando una funzione binaria.

- *show* - mostra su standard output (`stdout`) i primi n elementi del *Dataset*.

- *write* - scrive in output il *Dataset* con il formato e percorso specificati.

Questa distinzione è necessaria in quanto il paradigma di *Spark* costruisce un **piano logico** che rappresenta le computazioni necessarie per restituire i dati di un *Dataset*. Infatti, i *Dataset* hanno un comportamento *lazy*: le computazioni vengono eseguite in parallelo a *runtime* solamente alla chiamata di un'azione (figura **??**). Quando viene invocata un'azione, l'ottimizzatore di *Spark* ottimizza il piano logico e produce un **piano fisico** per eseguire le query efficientemente in maniera parallela e distribuita [**spark-dataset**]. Questa distinzione permette di rendere Spark estremamente dichiarativo: le sole trasformazioni permettono di gestire enormi moli di dati senza che queste vengano effettivamente eseguite a runtime nel momento in cui si dichiarano.

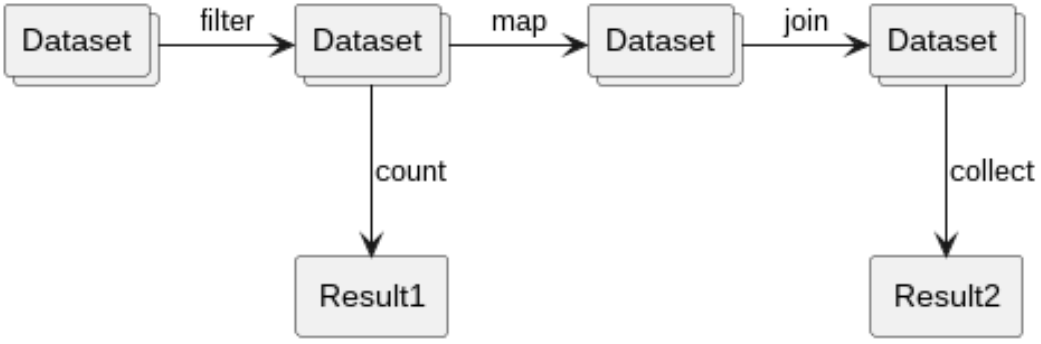

Figura 2.2: Esempio di esecuzione di trasformazioni e azioni su *Dataset*.

In più, la gestione dei tipi di dati dei *Dataset* permette di individuare eventuali errori nel piano logico senza dover aspettare l'effettiva esecuzione di quello fisico; ad esempio, l'invocazione di una *select* su un determinato campo può specificare che tale campo non esiste nel *Dataset*; oppure, una funzione di aggregazione per somma può restituire un errore logico se viene effettuata su una colonna di stringhe.

Inoltre, è possibile salvare in memoria il contenuto di un *Dataset* grazie al metodo `Dataset.cache()`. Ciò può essere estremamente utile nel caso in cui vengano effettuate più computazioni su uno stesso *Dataset*: mantenerne il contenuto in memoria permette di risparmiare tempo d'esecuzione del *job* per eseguire ulteriori *job* sullo stesso cluster. In aggiunta, è considerata come una operazione *lazy*, cioè che le informazioni del *Dataset* non vengono effettivamente rese persistenti fino all'invocazione di un'azione.

2.1.3 L'operazione di *join*

La trasformazione di *join* è una delle più importanti poichè permette di combinare informazioni contenute in diversi *Dataset* con una specifica *join condition*. Quando si applicano queste trasformazioni bisogna tener conto delle moli di dati che vengono **combinati in maniera distribuita**: spesso si può incorrere in problemi di prestazioni dovuti ad un utilizzo non corretto delle trasformazioni di *join*. Ad ogni modo, le trasformazioni di *join* sono ottimizzate da Spark di default, ma questo non significa che non vada prestata attenzione nel loro utilizzo [**spark-join**]. Spark supporta tutti i principali tipi di *join*

di SQL; di seguito vengono elencati e brevemente descritti prendendo come riferimento i *DataFrame*:

- **Inner Join** (figura **??**) - Corrisponde alla *join* di default. Viene utilizzata per combinare due *DataFrame* considerando delle colonne chiave. Laddove non vi sia corrispondenza tra le chiavi, le righe vengono rimosse da entrambi i *DataFrame*. Corrisponde all'individuazione dell'intersezione di due insiemi di dati, dove l'intersezione coincide con l'uguaglianza di determinati campi.

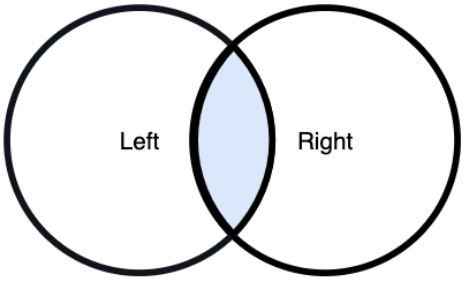

Figura 2.3: Esempio figurato di una *inner join* tra due *Dataset*: colonne di entrambi e righe che soddisfano la *join condition*.

- **Full Outer Join** (figura **??**) - Restituisce le righe di entrambi i *DataFrame*, lasciando il valore `null` nelle rispettive colonne quando le chiavi non corrispondono.

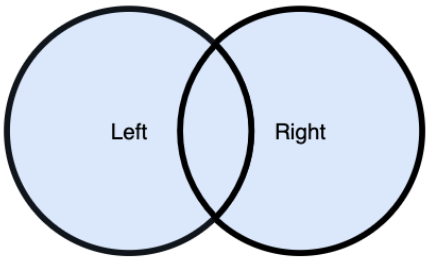

Figura 2.4: Esempio figurato di una *full outer join* tra due *Dataset*: righe e colonne di entrambi.

- **Left Outer Join** (figura **??**) - Restituisce tutte le righe del *DataFrame* di sinistra, a prescindere dalla corrispondenza delle chiavi. Laddove non vi sia corrispondenza tra le chiavi, le colonne del *DataFrame* di destra vengono valorizzate con `null`. Viene utilizzata quando non si vuole alterare la dimensione di un *DataFrame*, bensì si vuole arricchire di informazioni se presenti all'interno di un altro (quello di destra).

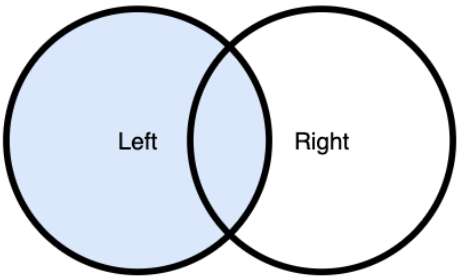

Figura 2.5: Esempio figurato di una *left outer join* tra due *Dataset*: colonne di entrambi, righe di quello di sinistra.

- **Right Outer Join** (figura **??**) - Corrisponde al contrario della *left outer join*. Restituisce tutte le righe del *DataFrame* di destra, valorizzando con `null` le colonne di quello di sinistra laddove le chiavi non corrispondano. Equivale ad effettuare una *left outer join* invertendo i *DataFrame*, ma viene comunque messa a disposizione per comodità.

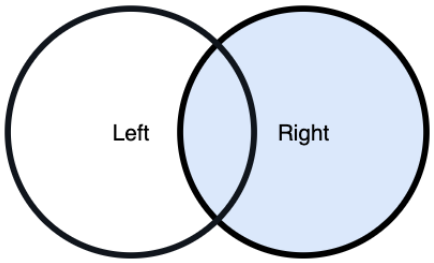

Figura 2.6: Esempio figurato di una *right outer join* tra due *Dataset*: colonne di entrambi, righe di quello di destra.

- **Left Semi Join** (figura **??**) - Equivale alla *inner join*, tuttavia ignora le colonne del *DataFrame* di destra. Più semplicemente, questa *join* restituisce le righe del *DataFrame* di sinistra quando si ha una corrispondenza delle chiavi. Corrisponde ad una *inner join* seguita da una *select* dei campi del *DataFrame* di sinistra.

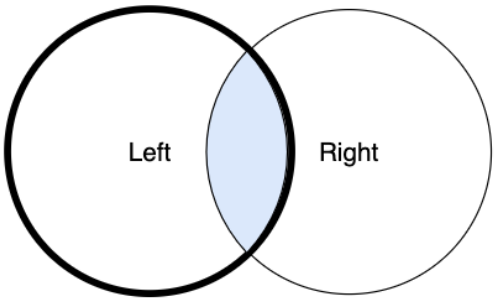

Figura 2.7: Esempio figurato di una *left semi join* tra due *Dataset*: colonne di quello di sinistra, righe che soddisfano la *join condition*.

- **Left Anti Join** (figura **??**) - Corrisponde all'opposto della *left semi join*. Restituisce le righe del *DataFrame* di sinistra che non hanno una corrispondenza con quello di destra.

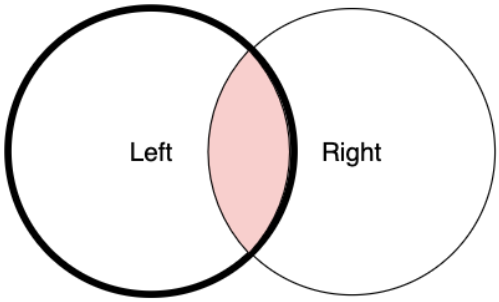

Figura 2.8: Esempio figurato di una *left anti join* tra due *Dataset*: colonne di quello di sinistra, righe che non soddisfano la *join condition*.

2.2 Il linguaggio: Scala

Il linguaggio di implementazione del progetto è Scala2. La scelta del linguaggio è dovuta ad una dipendenza di Spark. Le API dei moduli di Spark sono disponibili per diversi linguaggi; tra i più utilizzati si trovano *Python* e *Scala*. Inizialmente, i progetti Spark in Prometeia utilizzavano *Python* come linguaggio d'implementazione. Tuttavia, si è notato che un medesimo programma scritto in *Scala* presentava prestazioni nettamente migliori, quindi si è scelto di passare a tale linguaggio. Questa differenza di prestazioni è dovuta all'architettura stessa di Spark descritta nella sezione **??**. Nello specifico, i nodi *worker* istanziano una **Java Virtual Machine** su cui eseguire i vari task di Spark. Quindi, quando devono eseguire un piano fisico relativo ad un *Dataset* di oggetti *Python*, è necessario convertire tali oggetti in *Java*. Questo passaggio invece non è necessario quando i *Dataset* sono implementati in *Scala* per via dell'interoperabilità con *Java*. Di conseguenza, è stato scelto di implementare i nuovi progetti Spark in *Scala*.

In più, *Scala* porta con sè ulteriori vantaggi, tra i quali:

- Scalabilità - Anche il nome stesso suggerisce uno dei grandi vantaggi del linguaggio. Infatti, *Scala* è una abbreviazione di **Scalable Language**, stando a indicare come questi sia utilizzabile per costruire sistemi scalabili, anche di grandi dimensioni.

- Supporto degli ambienti di sviluppo - Diversi ambienti di sviluppo supportano questo linguaggio. In particolare, l'IDE (*Integrated Development Environment*) *IntelliJ Idea* fornisce una serie di funzionalità che migliorano l'esperienza di sviluppo; ad esempio, una chiara *syntax highlighting* e un supporto per la *type inference* che permette di intercettare errori a *compile time*.

- Sintassi semplificata - É un linguaggio più conciso rispetto a *Java*; infatti, spesso molteplici righe di codice *Java* possono essere riassunte in una sola in *Scala*. Basti pensare che tutte le operazioni degli `Stream` in *Java* sono invocabili da una qualsiasi *collection* di *Scala*.

- Linguaggio funzionale - *Scala* è un linguaggio che supporta l'**higher-order**. Ciò significa che le funzioni possono essere passate come argomenti ad

altre funzioni e utilizzate come valori. Questa caratteristica può migliorare la qualità del codice prodotto dagli sviluppatori, così come può facilitare la scrittura di algoritmi e procedure.

- Interoperabilità con Java - Come sopra anticipato è molto facile far cooperare codice *Java* e *Scala*. Questo può essere d'aiuto nel caso di dipendenze da librerie o da altre componenti sviluppate in precedenza che diventano quindi riutilizzabili senza dover essere riadattate.

- Pattern Matching - Un costrutto molto utile che può essere visto come lo `switch` di *Java* più potente. Può essere utilizzato per controllare il contenuto di una variabile, il tipo dinamico di un'istanza a *runtime*, iterare liste e molto altro.

2.3 Il cloud: Amazon Web Services

L'*Infrastructure as a Service* (*IaaS*) è una tipologia di *Cloud Computing* basato sul consumo, come servizio, di risorse hardware. Server virtuali, potenza di calcolo, storage, reti vengono messi a disposizione per essere utilizzati senza necessariamente dover affrontare costi di acquisto dell'hardware stesso. In questo modo, chi adopera dell'infrastruttura ottiene il totale controllo delle risorse a sua disposizione. Uno dei vantaggi maggiori rispetto all'acquisto diretto delle risorse è quello della manutenzione: chi utilizza l'infrastruttura *in cloud* non si deve preoccupare di fare manutenzione o rinnovare l'hardware; si limita semplicemente ad utilizzare un servizio scalabile e affidabile senza preoccuparsi dei meccanismi di gestione interna.

Invece, il *Platform as a Service* (*PaaS*) rappresenta una tipologia di servizio fornito del tutto differente, anche se per diversi aspetti simile alle *IaaS*. Ad essere fornito come servizio, in questo caso, non è solo l'hardware, ma anche la piattaforma che permette di usufruire di un insieme di funzionalità le quali forniscono storage, reti, macchine virtuali, deployment, bilanciamento del carico (*load balancing*) e altro ancora. A differenza delle *IaaS*, il meccanismo di queste funzionalità viene spesso oscurato: nelle *IaaS* si ha pieno controllo delle funzionalità a disposizione, mentre nelle *PaaS* no. Il vantaggio per l'utente è quello di concentrarsi solo ed esclusivamente sullo sviluppo delle applicazioni,

e non perdersi nell'analisi di problematiche legate all'ambiente in cui queste devono essere distribuite, ottenendo, contestualmente dalla piattaforma, la scalabilità e l'affidabilità necessaria. Inoltre, come per le *IaaS*, l'utente non deve preoccuparsi di aggiornare i sistemi operativi: viene tutto gestito dinamicamente e in modo del tutto automatico dalla piattaforma. Se in determinati periodi si dovessero verificare carichi di picco, la piattaforma deve essere in grado di adeguare la propria struttura per rispondere alle nuove esigenze, anche se temporanee.

Tra le diverse tecnologie di *Cloud Computing*, Prometeia fa anche uso di *Amazon Web Services* (AWS) per la realizzazione di sistemi e servizi con supporto in *cloud*. Tra i diversi servizi che AWS mette a disposizione, durante il tirocinio sono stati utilizzati i seguenti:

- **S3** - *Amazon Simple Storage Service* è un servizio di archiviazione di risorse. Offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. I clienti di tutte le entità e settori possono archiviare e proteggere qualsiasi quantità di dati per qualsiasi caso d'uso, come *data lake*, applicazioni native per il cloud e applicativi *mobile*. Con classi di archiviazione convenienti e funzionalità di gestione di facile utilizzo, è possibile ottimizzare i costi, organizzare i dati e configurare controlli di accesso ottimali per soddisfare specifici requisiti aziendali, organizzativi e di conformità [**aws-s3**].

- **Glue** - *AWS Glue* è un servizio di integrazione dei dati serverless che facilita la scoperta, la preparazione, lo spostamento e l'integrazione dei dati da più origini per l'analisi e lo sviluppo di applicazioni [**aws-glue**]. *AWS Glue* può eseguire i processi di estrazione, trasformazione e caricamento (ETL) non appena arrivano nuovi dati. Ad esempio, si può configurare per eseguire il processo ETL non appena i nuovi dati diventano disponibili su Amazon S3 (vedi figura **??**).

- **DynamoDB** - *Amazon DynamoDB* è un database NoSQL serverless e completamente gestito, progettato per eseguire applicazioni ad alte prestazioni su qualsiasi scala. Offre sicurezza integrata, backup continui, repliche multi-regione automatizzate, caching interno alla memoria e strumenti di importazione ed esportazione dei dati [**aws-dynamo**].

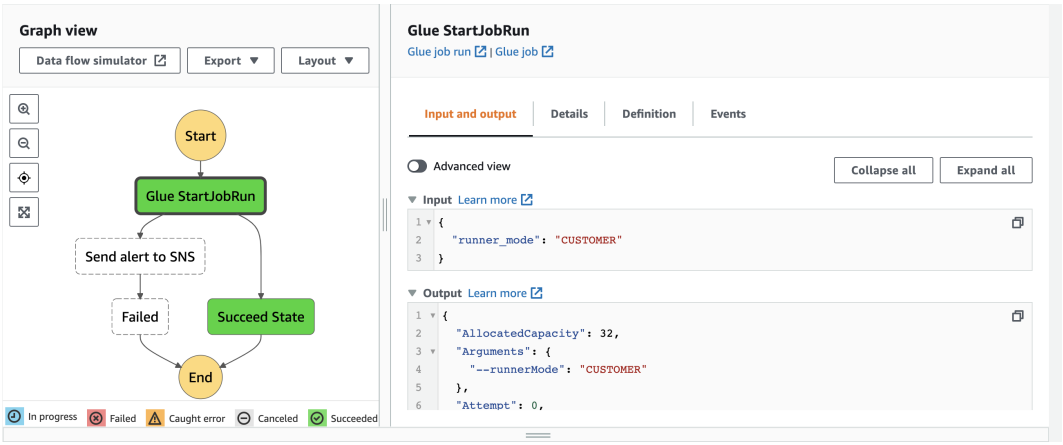

Figura 2.9: Esempio di *Step Function* che esegue un job Glue con diversi parametri.

- **Step Functions** - *AWS Step Functions* è un servizio che offre un flusso di lavoro visivo che permette di utilizzare ulteriori servizi AWS per costruire applicazioni distribuite, automatizzare processi e orchestrare microservizi [**aws-step**]. Ad esempio, offre la possibilità di combinare in sequenza o in parallelo diversi job di AWS Glue, creando una vera e propria macchina a stati con diversi percorsi a seconda del successo dei processi che la compongono (la figura **??** ne è un esempio).

- **SecretsManager** - *Amazon Secrets Manager* permette di modificare periodicamente, gestire e recuperare credenziali di database, chiavi API e altre chiavi segrete in tutto il loro ciclo di vita [**aws-secrets-manager**].

- **Athena** - *Amazon Athena* è un servizio di analisi interattivo serverless basato su framework open source, che supporta formati di file e tabelle aperte. Fornisce un modo semplificato e flessibile per analizzare petabyte di dati dove risiede. Permette di analizzare i dati o creare applicazioni da bucket Amazon Simple Storage Service (S3) e oltre 25 origini dei dati, incluse origini dei dati on-premise o altri sistemi cloud utilizzando SQL o Python. Athena è basato su motori *Trino* e *Presto* open source e framework Apache Spark, senza necessità di provisioning o configurazione [**aws-athena**].

- **OpenSearch** - Consiste in una suite di analisi dei dati e ricerca distribuita, controllata dalla community, con licenza Apache 2.0 e open source utilizzata per un'ampia gamma di casi d'uso come il monitoraggio delle applicazioni in tempo reale e l'analisi dei dati di registro. OpenSearch fornisce un sistema altamente scalabile per fornire accesso e risposta rapidi a grandi volumi di dati con uno strumento di visualizzazione integrato, *OpenSearch Dashboards*, che semplifica l'esplorazione dei dati da parte degli utenti. Si basa sulla libreria di ricerca Apache Lucene e supporta una serie di funzionalità di ricerca e analisi dei dati come la ricerca *k-nearest neighbors* (KNN), SQL, Anomaly Detection, Machine Learning Commons, Trace Analytics, la ricerca full-text e altro ancora [**aws-open-search**].

Il principale svantaggio di *IaaS* e *PaaS*, e in generale di tutte le tecnologie *as a Service*, è il rischio di **lock in**. I fornitori come *Amazon* offrono un servizio che obbliga i fruitori ad adoperare tecnologie specifiche. Sebbene sia un vantaggio per chi offre i servizi, questo potrebbe invece risultare problematico per chi li compra. La scelta di un *service provider* in fase di progettazione potrebbe causare danni economici anche notevoli in fase di sviluppo nel caso in cui questo non soddisfi a pieno le aspettative del fruitore. Da ciò si deduce quanto sia importante essere in grado di migrare da un *cloud provider* a un altro; questa possibilità dipende fortemente dalla portabilità delle applicazioni sviluppate e dei dati mantenuti in *cloud*.

L'utilizzo di AWS come *PaaS* permette di soddisfare il requisito principale del progetto, ovvero l'esecuzione di un ETL sulla base di dati ricevuti. Come discusso nella sezione **??**, la ricezione di flussi può avvenire per intero o parzialmente, cioè storicizzati. I servizi di AWS possono essere integrati tra di loro per customizzare la gestione differente di questi flussi: ad esempio, si può configurare *Glue* per eseguire l'ETL al caricamento di nuovi dati direttamente su *S3*.

Figura 2.10: Integrazione dei serivizi di AWS per realizzare un ETL basato su eventi [**aws-glue**].

Capitolo 3

Progettazione dell'architettura

In questo capitolo vengono descritti i sistemi realizzati dal punto di vista architetturale. Innanzitutto, verrà descritto l'ETL realizzato per il calcolo dei rendimenti e come è stata iniziata l'integrazione del nuovo PFP all'interno del progetto. Poi, verrà analizzata la componente realizzata per il prodotto di Baseline, spiegando quanto è stato fatto durante il tirocinio e quanto è stato progettato per i periodi successivi. Infine, verrà descritta l'infrastruttura e le tecnologie utilizzate per l'hosting di risorse e programmi.

3.1 ETL dei rendimenti e PFP

Al momento dell'inizio del tirocinio curriculare, il progetto era già stato iniziato. Di conseguenza, il lavoro svolto si limita all'implementazione del progetto già pianificato. In questa sezione viene descritta l'architettura del sistema realizzato specificando quali cambiamenti vi sono stati rispetto alla fase di progettazione.

Come descritto nell'introduzione alla problematica (sezione **??**), l'integrazione di dati è un processo molto complesso ed estremamente personalizzato; in questo senso, non vi è uno *standard de facto* da seguire per progettare l'*ETL* aziendale, tuttavia esiste una sorta di percorso che indica il modo in cui sia da effettuare l'integrazione. Tale percorso considera i seguenti aspetti:

- I dati derivanti da diverse sorgenti sono dotati di schemi differenti. Ogni schema può presentare delle informazioni in comune con gli altri, ma

non necessariamente con la stessa nomenclatura. Inoltre, bisogna tener conto che dati di sorgenti diverse possono essere rappresentati con formati diversi: ad esempio, due documenti in relazione tra loro possono avere formato *json* e *csv*. Diventa quindi essenziale che la prima fase dell'*ETL* individui una procedura per uniformare i formati e i riferimenti a campi in documenti correlati tra loro.

- Il processo di integrazione di dati deve comprendere le seguenti fasi:

  1. **Raccolta** - i dati devono opportunamente essere reperiti e inseriti in un unico punto di raccolta in modo tale da facilitare l'accesso ai flussi per l'integrazione.
  2. **Omogeneizzazione** - i flussi devono essere letti e uniformati per quanto riguarda il formato in cui si effettueranno le successive funzioni operative.
  3. **Validazione** - è necessario individuare i record che non sono schematicamente corretti, cioè che non rispettano lo schema e che qunidi sono da scartare (ad esempio, record con campi obbligatori non valorizzati).
  4. **Pulizia** - è necessario individuare i record che non sono semanticamente corretti, cioè non conformi al dominio trattato e che quindi sono da scartare (ad esempio, un record in cui una persona fisica è dotata di età negativa).
  5. **Valorizzazione** - l'assegnazione di valori a campi non valorizzati.
  6. **Trasformazione** - l'applicazione di modifiche ai dati, quali l'aggiunta o la rimozione di campi oppure la semplice trasformazione di valori associati.
  7. **Scrittura** - l'output prodotto dalle fasi precedenti deve essere riscritto in un apposito repository tenendo conto dei diversi casi d'uso del medesimo.

- I dati risultanti dall'integrazione devono avere lo stesso formato, non necessariamente corrispondente a quello di input. Spesso si utilizza un formato che favorisca le prestazioni per i vari casi d'uso delle informazioni prodotte.

Per quanto riguarda l'architettura del progetto definita da Prometeia, ci si è attenuti ad uno standard aziendale. Tenendo conto delle *best practices* sopra indicate, Prometeia ha individuato un compromesso tra la struttura ideale e i vincoli aziendali dati dall'obbligo di utilizzo di alcune tecnologie. Nello specifico, per questo progetto è stato utilizzato *Amazon Web Services* come *cloud service provider*; di conseguenza, sono stati sfruttati molteplici servizi in *cloud* per diversi scopi, quali hosting, testing, storing e altri.

Le componenti da realizzare per implementare l'ETL sono state individuate sulla base delle necessità del cliente, in termini di risorse e prestazioni, ovvero sulla base di quali dati sono richiesti e dove questi vadano memorizzati e di quali tempistiche d'esecuzione sono richieste. Ogni componente è stata quindi incapsulata in un *job* di *AWS Glue*, un servizio di integrazione dei dati serverless [**aws-glue**] utile per schedulare l'esecuzione di programmi. Le componenti hanno una dipendenza quasi del tutto sequenziale: nello specifico, i batch vengono eseguiti da una **Step Function** di *Amazon Web Services*. Quest'ultima permette di definire la sequenza di job da eseguire, specificando dipendenze temporali e anche quali batch possono essere lanciati in parallelo. Il diagramma di attività in figura **??** rappresenta la sequenza dei job e le dipendenze tra loro.

3.1.1 Validazione

Il job di validazione si occupa di leggere tutti i flussi in input nel formato *csv*, validarli, applicare delle trasformazioni e riscriverli in formato *parquet* [**parquet**]. Per poter validare i dati in input, è necessaria la configurazione di un tracciato che determini quale schema ci si aspetta dai flussi. La validazione comprende diversi controlli per verificare quali record siano da scartare e quali possano passare alle fasi successive. I controlli necessari sono:

- Chiave primaria duplicata - la chiave primaria determina l'identificatore di un record; di conseguenza, i record con stessa chiave primaria sono da scartare in quanto non si sa quale sia corretto.

- Chiave esterna non censita - per i flussi che hanno una chiave esterna è necessario controllare che questa sia presente nel flusso di riferimento. Ad esempio, se esiste un rapporto su un determinato cliente, nel flusso

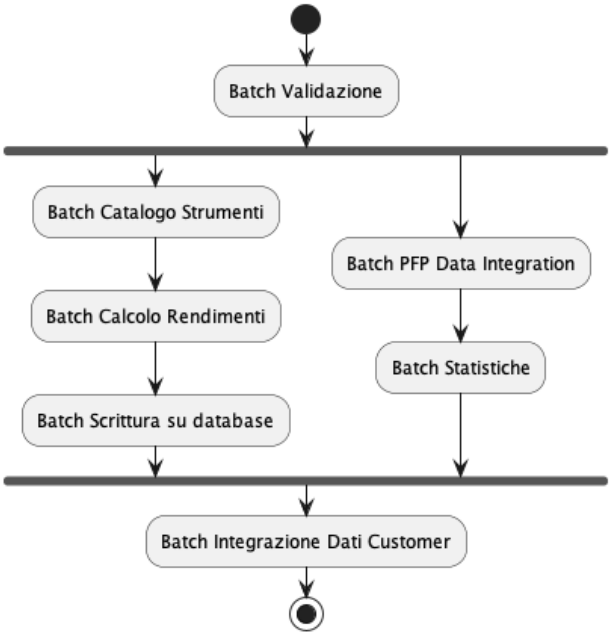

Figura 3.1: Rappresentazione grafica della *step function* che esegue l'ETL del progetto.

relativo ai rapporti sarà presente la chiave del flusso dei clienti. Bisogna verificare se effettivamente esiste un cliente con quella chiave nell'apposito flusso, altrimenti il record del rapporto sarà da scartare.

- Campi obbligatori - nel tracciato deve essere indicato per ciascun campo se questo può essere `nullable` o meno. Di conseguenza, è necessario scartare tutti i record che presentano campi valorizzati a `null`, ma che sono considerati obbligatori.

- Tipi di dato - nel formato *csv*, tutti i dati sono salvati come sequenza di caratteri; tuttavia, gli schemi successivi possono richiedere dei dati con tipi specifici, quali intero, booleano e altri. Diventa quindi necessario verificare che possa essere effettuato il *cast* dei dati presenti nei flussi come indicato nel tracciato per ciascun campo.

Il caricamento in memoria dei flussi deve avvenire in modo efficiente per garantire le migliori prestazioni. L'ordine con cui vengono caricati i flussi influisce sulle prestazioni e sul risultato della validazione in quanto questi

controlli, assieme all'*enrichment* dei dati, vengono fatti in sequenza sulle tabelle. Ad esempio, per controllare il censimento di un record con chiave esterna, è necessario che il flusso con la *foreign key* sia già stato importato. Quindi, per ovviare a questo problema, si affronta una fase di caricamento che costruisce un ordine specifico sulla base del modello dei flussi. Nello specifico si costruisce un **direct acyclic graph** (DAG) rappresentante le dipendenze dei flussi tra loro, dove una dipendenza può derivare da una chiave esterna o da altre necessità (come l'utilizzo di un flusso *ad hoc* per arricchire i dati di un altro). A partire dal DAG, si deduce la sequenza con cui vengono quindi ordinati i flussi; questo rimane poi l'ordine con cui vengono mantenuti e con cui avvengono tutte i controlli e le trasformazioni. La figura **??** mostra il DAG semplificato del modello trattato (non sono stati riportati tutti i campi in quanto non necessari per l'apprendimento di questa parte, così come non sono state riportate le chiavi primarie dei flussi per intero per rendere la figura più compatta).

Inoltre, i controlli producono due output: uno contenente i record validati, e l'altro quelli scartati. Entrambi vengono salvati in formato *parquet* all'interno di due repository sulla base della natura del flusso: un flusso può essere ricevuto dal cliente *in full* oppure *in delta*, dove la prima opzione indica un flusso che viene mandato una volta sola poichè il suo contenuto non cambia, mentre la seconda indica flussi che continuano ad aumentare di volume e che vengono quindi inviati continuamente. Per quanto riguarda i flussi *in delta*, è stato pattuito col cliente che invii solo i nuovi record, in quanto non avrebbe senso processare informazioni che sono già state elaborate dall'*ETL*. Quindi, per distinguere le due tipologie di flusso, queste vengono salvate in due repository separate: `output/output` per i flussi *in full* e `output/store` per quelli *in delta*. Invece, per distinguere flussi validi e non, gli scarti vengono salvati in un repository `output/output/errors` oppure `output/store/errors` a seconda del flusso; inoltre, il cliente può accedere a questi repository, in modo tale da non dover comunicare errori nei flussi ogniqualvolta si verifichino, come da requisito (vedi sezione **??**). Il concetto appena descritto è rappresentato nella figura **??**.

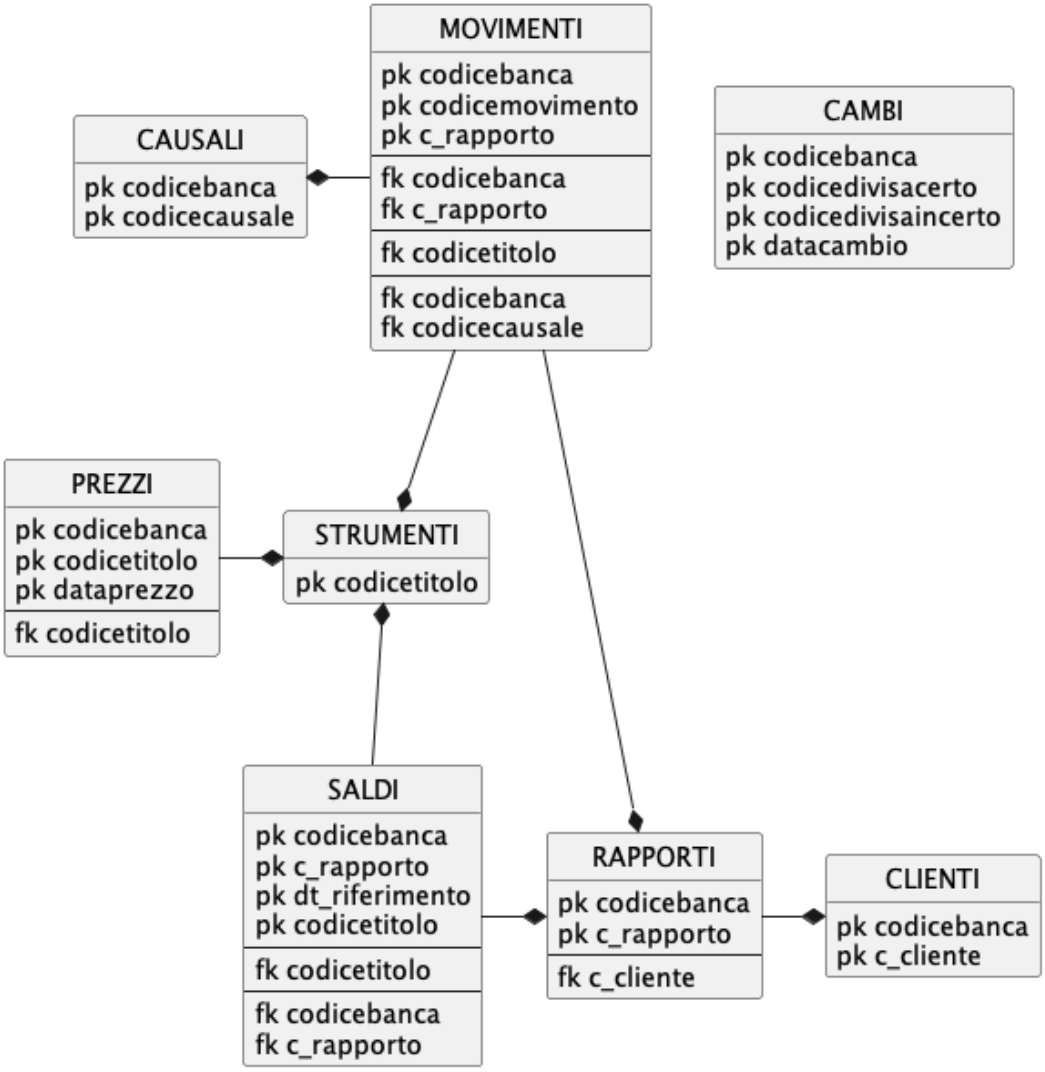

Figura 3.2: Grafo diretto aciclico delle dipendenza tra i flussi.

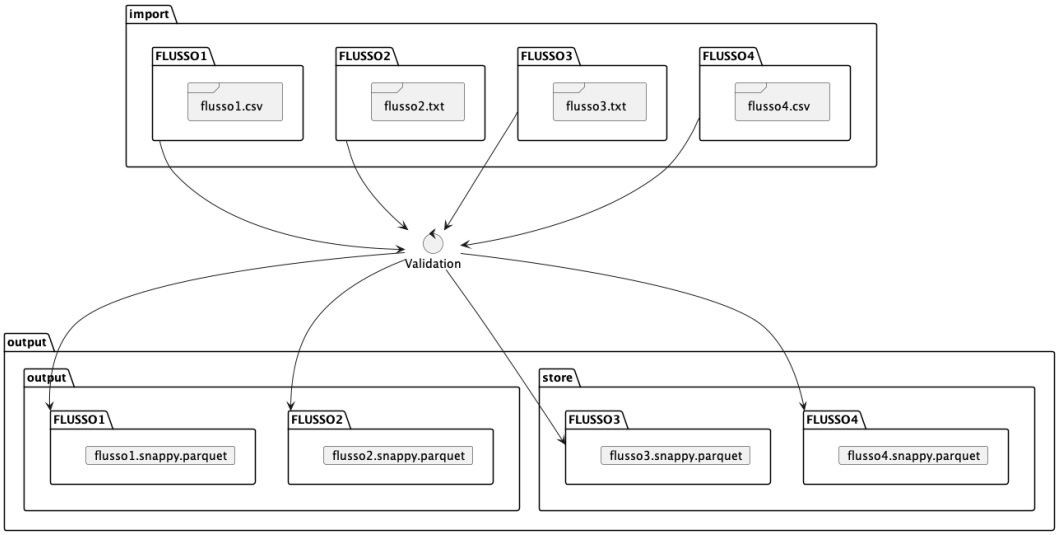

Figura 3.3: Processo di salvataggio dei file *parquet*, distinguendo tra flussi *in full* e *in delta*.

3.1.2 Catalogo Strumenti

Il job di integrazione degli strumenti finanziari, soprannominato *cat*, è necessario per mantenere integrato il contenuto del flusso riguardante i prodotti finanziari, uno dei flussi *in full* più importanti. Questo infatti contiene i riferimenti a tutti gli strumenti finanziari, ed è utilizzato da molteplici altri flussi per ottenere informazioni su di essi. Quindi, affinchè quest'ultimo non contenga campi vuoti o campi superflui poiché non utilizzati, è necessario un job che si occupi appositamente della sua pulizia e integrazione. In particolare, il job in questione prende in input i flussi in formato *parquet* prodotti dal job di validazione e scrive su un indice **OpenSearch** in modo che diversi endpoint possano reperire queste informazioni tramite *API REST*.

Questo batch ha il compito di integrare diverse informazioni presenti in flussi diversi all'interno di un unico indice. In particolare, per ciascun strumento finanziario vanno reperite diverse informazioni presenti in flussi separati che contengono un riferimento al titolo dello strumento. Quindi, la struttura di tale batch deve permettere di prendere in input solo determinati flussi, cioè quelli necessari ad integrare le informazioni sugli strumenti. Una volta prelevate tali informazioni, può avvenire l'integrazione: si può affermare che questa parte, se

ben strutturata e priva di errori, non genererà errori dovuti ad incoerenze di schema come chiavi esterne non censite, dato che tali controlli sono già avvenuti nel batch della validazione descritto nella sezione **??**. In seguito all'integrazione, gli strumenti finanziari saranno quindi completi e potranno essere generati i due principali output di questo batch:

- Indicizzazione dei dati sugli strumenti e caricamento su *OpenSearch*.
- Scrittura dei dati sugli strumenti in formato *parquet* in una directory separata rispetto a quella del batch precedente.

La duplicazione dell'output è dovuta ai diversi casi d'uso di questi: l'indice su *OpenSearch* è storicamente utilizzato da diversi endpoint tramite *API REST*; risultano quindi necessari la scrittura e l'aggiornamento di tale indice. Invece, la scrittura delle medesime informazioni in *cloud* in formato *parquet* avviene per comodità: infatti, potrebbe risultare scomodo per i batch successivi reintegrare determinate informazioni relative ad uno strumento finanziario quando tale operazione è già avvenuta. Quindi, semplicemente si mantengono in directory separate il flusso ricavato dalla validazione e quello ottenuto dal catalogo.

3.1.3 Calcolo Rendimenti

Il job dei rendimenti si occupa di calcolare i rendimenti per ciascun cliente su diverse finestre temporali (vedi sezione **??**). Tale job prende in input i flussi in formato *parquet* prodotti dalla validazione e li integra con i rendimenti calcolati sulla base delle finestre temporali richieste (che quindi costituiscono un parametro del job). I risultati dei rendimenti, anche detti *return items*, vengono ulteriormente scritti in formato *parquet* in un repository `output/returnitems`.

3.1.4 Scrittura su database

Un requisito importante è costituito dalla scrittura dei *return items* su un database *PostgreSQL*. Questa necessità è dovuta dall'esistenza di diverse componenti front end che reperiscono alcuni dati (non tutti) dei rendimenti direttamente da database *Postgres*. Di conseguenza, tale job si occupa di

rileggere i rendimenti in formato *parquet*, applicare alcune trasformazioni di pulizia e valorizzazione di campi aggiunti e scrivere il risultato su un database remoto. Tali operazioni venivano precedentemente effettuate all'interno del job di calcolo dei rendimenti; tuttavia, ci si è resi conto che la scrittura su database costituisce un collo di bottiglia piuttosto oneroso per quanto riguarda le prestazioni del job. Di conseguenza, è stato deciso di dedicare un batch apposta per la scrittura, in modo tale da poterlo eseguire in parallelo alle fasi successive.

Per poter accedere al database, sono necessarie delle credenziali e una serie di informazioni utili per stabilire la connessione. Queste informazioni costituiscono un *asset* da proteggere in quanto concedono l'accesso a dati dei clienti: risulta quindi necessario mantenerle all'interno di un segreto, cioè risorse protette sbloccabili solo tramite una chiave riservata. Pertanto, è stato deciso di fare uso di *AWS SecretsManager*. Infatti, tale servizio deserializza le informazioni contenute all'interno del segreto che non potrebbero essere mantenute nei sorgenti del componente, come *hostname* e *password* del database. Dopodiché procede con la scrittura di un sottoinsieme di informazioni sul database.

3.1.5 Data Integration PFP

L'integrazione dei dati del PFP è stata momentaneamente gestita in un batch separato da quello della validazione. In futuro, con grande probabilità, questa componente verrà fusa con quella di validazione. Questo batch ha lo scopo di reimplementare un modulo già presente e operativo in produzione, ma che necessita di una rivisitazione con nuove tecnologie per abbattere i lunghi tempi d'esecuzione. A livello architetturale, presenta le stesse caratteristiche del batch di validazione.

3.1.6 Statistiche

In preparazione per la fase finale, che prevede l'integrazione dei dati dei clienti finali della banca, è necessario reperire **massivamente** informazioni su di essi. Le chiamate massive, sono così dette in quanto elaborano una serie di dati per ciascun cliente, richiedendo quindi un certo ammontare di tempo; motivo per cui è stato ritenuto opportuno separarne l'esecuzione dai job

precedenti e successivi. Le informazioni in questione si ottengono chiamando degli *engine* che effettuano calcoli sulla base dei dati presentati per ciascun cliente. Affinchè queste informazioni possano essere reperite, è necessario calcolare delle statistiche su ciascun cliente che servono in input agli *engine*; questo batch si occupa di calcolare tali statistiche e riscrivere tutti i flussi necessari nel formato richiesto dalle chiamate massive.

3.1.7 Integrazione Dati *Customer*

La fase finale consiste nel reperimento di tutte le informazioni ottenute durante le fasi precedenti, l'integrazione opportuna e la trasformazione in formato *json* per poter scrivere il risultato su un indice **OpenSearch**. Per poterlo fare, è necessaria la rilettura dei *parquet* con i dati dei clienti e delle informazioni ottenute dalle chiamate massive. Una volta raccolte tutte le informazioni necessarie per il risultato finale, sarà sufficiente individuare un modo per filtrare i dati da scrivere su indice e un modo per uniformare tutto in formato *json*.

3.2 Baseline

Come anticipato nella sezione **??**, la parte del progetto Baseline realizzata durante il tirocinio è stata iniziata durante lo svolgimento dello stesso. Lo scopo di questo progetto è quello di realizzare un prodotto che riesca ad emulare tutti gli step di un tipico progetto di ETL come quello discusso nella sezione **??**. Le componenti da realizzare sono state individuate dagli analisti e dagli ingegneri di Prometeia osservando progetti con diverse caratteristiche in comune. L'esigenza di partire con questo progetto per Baseline è nata per via della necessità di avere un catalogo interno degli strumenti finanziari. Quindi, per la durata del tirocinio, l'avanzamento di questo progetto si è limitata al completamento del catalogo degli strumenti. Eventuali componenti aggiuntive, come quella del calcolo dei rendimenti, verranno introdotte in momenti successivi.

Queste componenti hanno una dipendenza sequenziale (figura **??**), in quanto la corretta esecuzione del secondo dipende fortemente dal successo del primo. Alcuni casi d'uso del catalogo degli strumenti sono quello dell'analisi dati e

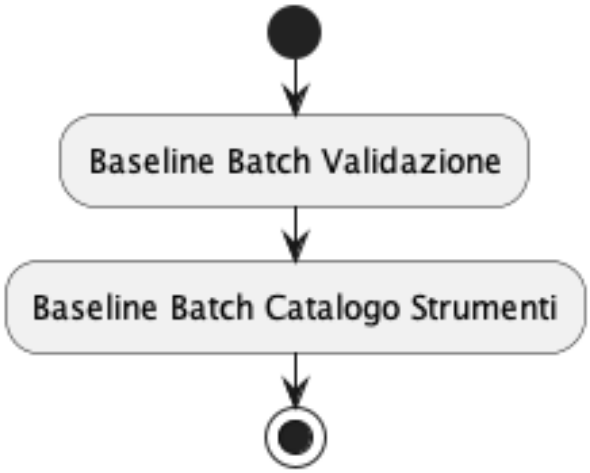

Figura 3.4: Rappresentazione grafica della *step function* che esegue i batch di Baseline.

quello del reperimento di informazioni sfruttando l'indice *OpenSearch* come *Software as a Service*.

3.2.1 Validazione

Ha lo scopo di caricare i flussi necessari per ottenere tutte le informazioni del catalogo degli strumenti. Nello specifico, si compone di due parti: una prima parte che carica diversi flussi unicamente relativi agli strumenti in formato *csv* e produce in output un flusso nel medesimo formato combinando le informazioni più adatte di ciascuno di quelli di input; la seconda parte esegue la validazione di tutti i flussi necessari per integrare il catalogo degli strumenti, compreso quello prodotto dallo step precedente. L'output della validazione corrisponde alla produzione di flussi in formato *parquet*.

3.2.2 Catalogo Strumenti

Questa fase ha lo scopo di leggere in input i file *parquet* prodotti dallo step di validazione e integrare tutte le informazioni per ciascuno strumento finanziario. Idealmente, ogni flusso validato contiene una chiave con il codice del titolo, che rappresenta la chiave degli strumenti. Quindi, un compito importante di

questo *job* è quello di raggruppare ciascun record per il codice del titolo ed effettuare un'operazione di *join* per ogni strumento con tutti i flussi necessari. Una volta integrate tutte le informazioni, il risultato ottenuto viene scritto su un indice *OpenSearch*.

3.3 Infrastruttura

Come anticipato, Prometeia utilizza le tecnologie di *Amazon Web Services* per avere supporto in *cloud*. Il *cloud service provider* di Amazon mette a disposizione una serie di servizi che possono essere sfruttati per diversi scopi, tra cui hosting e storing. Questo capitolo riassume l'architettura utilizzata per mantenere le risorse in *cloud* e permettere la loro interazione.

3.3.1 Hosting

I progetti sviluppati sono stati mantenuti in dei repository *GitHub*. In particolare, Prometeia ha diverse organizzazioni sul sito citato; generalmente, una per ciascuna area di competenza. Il tirocinio si è svolto all'interno dell'area di **WAM** (*Wealth and Asset Management*), che coinvolge diversi team e riguarda lo sviluppo di sistemi per il benessere dei clienti delle banche. Ne consegue che l'organizzazione *GitHub* che ha mantenuto i repository dei progetti è stata *prometeia-wam*. All'interno di tale organizzazione sono stati creati diversi repository, diversificati per cliente e tecnologie utilizzate. Ad esempio, relativamente al progetto di questa tesi, sono stati creati, per i diversi progetti, i repository

- `<progetto>-scala-spark`,
- `<progetto>-product-catalog`,

dove il primo contiene codice *Scala* che sfrutta la tecnologia *Spark* per realizzare l'architettura descritta nela sezione **??**, mentre il secondo contiene esclusivamente il sorgente del batch *Catalogo Strumenti*. La diversificazione è stata fatta in quanto il secondo progetto è stato realizzato utilizzando *Python* per questioni di flessibilità del codice, poiché tale batch necessita di scrivere su un indice *OpenSearch*. La scelta del linguaggio *Python* per tale componente

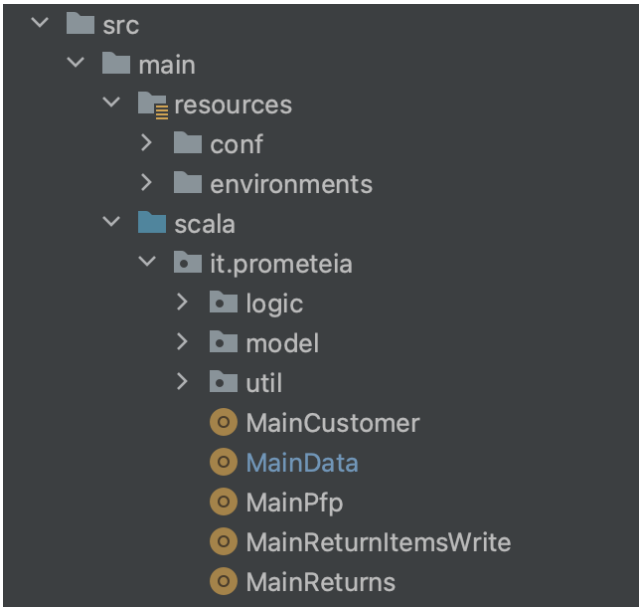

Figura 3.5: Programmi *main* del progetto *Scala* e i package contenuti nel repository di sviluppo.

è stata fatta per via dell'esistenza di una libreria che facilita la scrittura su indice: `opensearchpy`. I nomi dei repository sono stati rispettivamente `baseline-scala-spark` e `baseline-product-catalog` per quanto riguarda il progetto Baseline. Per il progetto esterno, realizzato per il cliente, è stato semplicemente inserito il nome dello stesso.

Tornando al primo repository, esso contiene la maggior parte del lavoro svolto. Ogni batch descritto nela sezione **??** è stato sviluppato all'interno del repository in questione, come programma *Scala*. In particolare, ogni batch viene inizializzato a partire da un *main* apposito che richiama risorse interne al progetto oppure di prodotto, quindi librerie esterne, ma comunque sviluppate internamente a Prometeia. Per dare un'idea più chiara della struttura interna al repository, la figura **??** mostra il contenuto del repository, senza andare nel dettaglio di ciascun package. Un altro dei vantaggi offerti da *GitHub* è quello delle **GitHub Actions**. Infatti, queste permettono di realizzare delle *pipeline* di *continuous integration* internamente ai repository. Tale servizio è stato sfruttato per eseguire il *deploy* di una versione funzionante dei programmi realizzati.

3.3.2 Deployment

Il *deployment* dei sistemi viene gestito in un contesto **multi-environment**, in cui ogni ambiente ha obiettivi precisi. Nello specifico, Prometeia ha individuato un insieme di ambienti sui quali operano diverse entità con diverse funzionalità:

- **Sviluppo** - Soprannominato *dev*, consiste nell'ambiente su cui vengono create e testate le applicazioni. Gli sviluppatori, seguendo determinati standard, costruiscono i sistemi in questo ambiente testandoli da diversi punti di vista. Questo ambiente non è visibile ai clienti, per cui eventuali errori sono accettabili. Non per niente, è l'ambiente in cui si verificano errori con maggiore frequenza; infatti, in fase di sviluppo le soluzioni non sono ancora sufficientemente stabili e necessitano di ulteriori test per individuare bug.

- **Staging** - Soprannominato *stag*, è l'ambiente in cui vengono trasferite le feature che si vogliono presentare al cliente tramite *demo*. Consiste nell'ambiente su cui gli analisti effettuano la maggior parte delle verifiche per individuare eventuali *bug* o errori da risolvere. L'incorrere di errori in questo ambiente è generalmente allarmante poiché significa aver tralasciato aspetti importanti in ambiente di sviluppo.

- **Pre-produzione** - Soprannominato *preprod*, è un ambiente condiviso con il cliente che quindi può impiegare analisti per ulteriori verifiche. Consiste in un'area in cui il cliente può effettivamente analizzare il lavoro svolto per verificare che coincida con i requisiti e le aspettative.

- **Produzione** - Soprannominato *prod*, è l'ambiente su cui viene effettuato il *deployment* delle applicazioni finali, le quali diventano quindi disponibili e utilizzabili dagli utenti. Gli errori in produzione sono inaccettabili, motivo per cui, quando si carica qualcosa su questo ambiente, bisogna agire con attenzione e cautela.

- **Ambiente Bolla** - Un ambiente aggiuntivo considerato una copia della produzione su cui effettuare dei test. Nello specifico, consiste in una replica dello stato dell'ambiente di produzione impiantato ad una determinata data. In questo modo, è possibile testare nuovi sviluppi prima di

renderli effettivi in produzione con le stesse condizioni, verificando che non avverranno errori al momento dell'*upgrade*.

Ad esclusione di *dev*, tutti gli altri ambienti vengono definiti **multitenant**. Infatti, i permessi di accesso a questi ambienti si dividono in due macrogruppi:

- `permessi-dev`, che permettono di accedere all'ambiente di sviluppo e sono concessi a tutte le risorse coinvolte nello sviluppo dei progetti;
- `permessi-mt`, che permettono di accedere a tutti gli ambienti oltre a quello di sviluppo; questi sono concessi solamente a risorse con alta responsabilità all'interno dei progetti.

Il *deployment* è gestito in modo automatico solamente in ambiente di sviluppo. In particolare, è stato scelto di caricare la versione aggiornata dei progetti solamente nel caso di *push* sul branch *develop*. Infatti, ogni progetto mantenuto su *GitHub* è dotato di un *workflow* che viene eseguito nei casi citati, simile a quello mostrato nel listato **??**. Nello specifico, questi flussi di *continuous integration* eseguono:

1. una *build* del repository con **Maven** come strumento di *build automation*,
2. l'autenticazione con credenziali di *Amazon Web Services*,
3. la copia della libreria prodotta e dei *main* su un bucket di S3.

In questo modo, eseguendo la pipeline, è possibile ottenere il *deploy* di una versione funzionante dei programmi realizzati. Una volta effettuato il *deploy* dei programmi, è possibile lanciarli manualmente e individualmente dal servizio *Glue*: quando l'intero sistema verrà rilasciato in produzione, questo verrà eseguito in sequenza con una determinata frequenza, ad esempio una volta al giorno; per ora, vengono eseguiti manualmente i singoli batch per produrre flussi in output e testare il risultato prodotto (il testing del sistema viene discusso nella sezione **??**).

3.3.3 Testing

La creazione di suite di test per i sistemi di ETL realizzati in Spark risulta essere problematica. Innanzitutto, la realizzazione di *Unit Test* per ogni

trasformazione sarebbe complessa: per via della natura dichiarativa delle trasformazioni di Spark (discussa nella sezione **??**), testare i *Dataset* risultanti consisterebbe nella semplice verifica delle dimensioni di questi, in termini di record, ma anche di campi o colonne. Inoltre, alcune feature non sarebbero testabili in quanto presenterebbero comportamenti diversi in locale o su un cluster. Diverse procedure in Spark presentano caratteristiche diverse a seconda di quanti nodi vengono utilizzati per il calcolo distribuito. Un esempio è la funzione colonna `monotonically_increasing_id`: quest'ultima genera numeri interi a 64 bit *monotonically increasing*, cioè crescenti, ma non necessariamente consecutivi. Quindi tale funzione riesce ad assegnare un numero unico e identificativo ad ogni record, ma la sua implementazione potrebbe essere male interpretata. Infatti, per ogni id, essa inserisce il numero identificatore della partizione corrente nei primi 31 bit, e il numero del record all'interno della partizione nei restanti 33 bit. Per esempio, supponendo di avere un *Dataset* con 2 partizioni, ognuna contente 3 record, la funzione `monotonically_increasing_id` restituirebbe i seguenti id:

$$0, 1, 2, 8589934592, 8589934593, 8589934594.$$

Si puo notare come tutti i numeri siano effettivamente unici. Tuttavia, non rappresentano il numero effettivo della riga all'interno dell'intero *Dataset*. Di fatto, ciò è valido solo per i primi 3 id, cioè quelli corrispondenti alla prima partizione. Questo perchè il numero della prima partizione corrisponde intuitivamente a 0, quindi l'id risultante corrisponde a quello del numero record inserito negli ultimi 33 bit. Come si può capire, questa funzione presenta comportamenti diversi a seconda del numero di nodi disponibili per un determinato *job*; di conseguenza, l'individuazione di *Unit Test* adatti per un *Dataset* che la utilizza risulterebbe difficile.

Per come è stata pensata l'infrastruttura dei sistemi, il momento adatto per eseguire delle ipotetiche suite di test sarebbe quello precedente al *deployment* con le *GitHub Actions*. Tuttavia, per i motivi sopra citati non sono stati inseriti controlli con test nella procedura di *deployment*. Quindi, per testare i sistemi realizzati sono stati utilizzati principalmente due metodi:

- Esecuzione dei batch in locale con un sottoinsieme di dati estremamente ridotto.

- Analisi dati tramite query sul servizio *AWS Athena* in seguito all'esecuzione dei job su *AWS Glue*.

A seconda delle problematiche da risolvere o delle feature da introdurre, sono stati utilizzati l'uno o l'altro approccio o entrambi.

Per quanto riguarda il primo approccio, è stato preso un sottoinsieme di record per ciascun flusso, facendo particolare attenzione che i record scelti restituiscano delle *join* non vuote. Quindi, per testare delle feature che coinvolgono l'operazione di *join* o altre trasformazioni, sono stati testati i sistemi eseguendoli in modalità di *debug*, e verificando che il risultato effettivo corrisponda a quello atteso, oltre che non si presentino errori a *runtime*. La riduzione dei flussi è stata fatta in quanto eseguire i batch in locale con i dati completi richiederebbe diverse ore per ciascun batch; mentre l'esecuzione effettiva sui cluster di AWS richiede pochi minuti.

Per quanto riguarda il secondo approccio, viene generalmente utilizzato dagli analisti per confermare la correttezza delle feature introdotte, ma viene anche utilizzato dagli sviluppatori per verificare casi specifici e delicati come quello della `monotonically_increasing_id` descritto sopra. In alcuni casi, l'esecuzione in locale dei batch non risulta sufficiente per verificare l'esattezza di una feature. Ad esempio, volendo testare dei comportamenti particolari dovuti alla natura distribuita di Spark oppure casi specifici come informazioni relative a uno strumento finanziario in particolare. Per questi casi quindi la prassi è quella di effettuare delle query su *AWS Athena* in seguito all'esecuzione dei job su *Glue*. Spesso, tali query vengono fornite a priori dagli analisti, i quali conoscono bene il risultato atteso e perciò sono anche qualificati per l'individuazione di queste.

Crawlers

Per comprendere il modo in cui vengono effettuati i test, è necessario citare il servizio *Crawlers* di AWS Glue. Tale servizio è in grado di sopperire alla problematica della lettura dei file in formato *parquet*: in particolare, come è stato anticipato nella sezione **??**, in seguito alla validazione i flussi vengono riscritti nel formato citato per garantire delle migliori prestazioni in lettura per i processi successivi. Tuttavia, la scelta di salvataggio dei dati in file di formato

binario ne complica il testing. La scelta migliore per poter effettuare agilmente delle query su di essi sarebbe quella di mantenerli all'interno di un database relazionale: infatti, all'interno dei database relazionali, la strutturazione dei contenuti è rigida; i dati vengono normalizzati e inseriti in delle tabelle, quindi salvati secondo un preciso schema. Lo schema definisce righe, colonne, indici, relazioni tra tabelle e ulteriori elementi e attua l'integrità referenziale nelle relazioni [**rdbms-vantaggi**]. Invece, nei file in formato *parquet*, così come in altri formati o altre tecnologie, i dati vengono salvati senza tabelle né schemi univoci. Nel caso corrente, la dipendenza dall'infrastruttura in *cloud* è stata prevalente rispetto alla comodità della strutturazione rigida dei database relazionali, quindi è stato adoperato il servizio dei *crawlers* di AWS Glue per risolvere tale problema. I *crawler* sono processi eseguibili che permettono di creare definizioni di tabelle a partire da file non strutturati. Di fatto, costituiscono il collegamento tra il *datastore* rappresentato dai bucket di S3 e il *query editor* di Amazon Athena. Possono essere configurati con dei parametri come il percorso dei file da analizzare e il loro formato. Quando eseguiti, questi vanno a leggere tutti i file nel formato e percorso specificati ed effettuano un'analisi che permette di dedurre lo schema relazionale tra di essi. A partire dallo schema, vengono creati dei metadati che puntano ai file *parquet*. In questo modo, Amazon Athena punta direttamente a questi metadati, garantendo la possibilità di effettuare analisi con delle query in linguaggio SQL. Quindi, i *crawler* vengono rieseguiti ogniqualvolta vi sia un cambiamento nello schema dei flussi.

Nel caso del progetto in questione, sono stati creati diversi *crawler* a seconda del diverso percorso dei file mantenuti sullo storage di S3:

- `input-crawler` - tale *crawler* permette di inferire lo schema relazionale di file in formato *csv* in input. Di fatto, la possibilità di effettuare query anche sui flussi ricevuti in input permette di confermare la correttezza di questi e di effettuare confronti per verificare la presenza di record prima e dopo l'esecuzione dei batch.

- `output-crawler` - tale *crawler* esamina i file in formato *parquet* esclusivamente di flussi *in full* poiché separati da quelli *in delta*.

- `scarti-crawler` - tale *crawler* analizza i record scartati. È utile per controllare il motivo degli scarti; infatti, quando un record viene scartato dalla validazione, al suo *DataFrame* viene aggiunta una colonna denominata `motivo_scarto` che ne contiene la causa.

- `store-crawler` - tale *crawler* analizza i record dei file storicizzati, cioè appartenenti a flussi *in delta*.

- `returns-crawler` - tale *crawler* esamina i file in formato *parquet* salvati dopo l'esecuzione del batch dei rendimenti.

3.4 Sviluppi futuri

In questa sezione si elencano i possibili sviluppi futuri dell'architettura e dell'infrastruttura, ideati durante il tirocinio a fronte delle possibilità offerte dalle tecnologie utilizzate.

3.4.1 Omogeneità di linguaggi

Uno sviluppo futuro programmato è quello di rimuovere completamente le dipendenze dalla libreria `opensearchpy` in modo da non avere componenti realizzate in Python. Come già anticipato, le componenti realizzate in Python tendono a rallentare le prestazioni: siccome Spark è realizzato in Scala, quando vengono utilizzate le API di Python la libreria deve necessariamente trasformare gli oggetti da un linguaggio all'altro. Questa trasformazione riduce esponenzialmente le prestazioni dei *job* allo scalare della quantità di dati in input. Di conseguenza, si considera opportuno rimuovere le dipendenze dalla libreria citata e migrare le componenti realizzate in Python al linguaggio Scala. La soluzione ideale sarebbe quella di realizzare a prodotto una logica che permetta di definire il *mapping* diretto a partire dal modello delle tabelle in Spark a quello degli indici di OpenSearch. Una prima idea da cui si potrebbe impostare tale logica è descritta nella sezione **??**.

3.4.2 Passaggi di feature a prodotto

Mano a mano che vengono realizzati componenti di ETL per diversi progetti, ci si rende conto che alcune feature risultano essere estremamente simili. In particolare, l'obiettivo è quello di non duplicare logiche e ottenere uno standard riapplicabile nel momento in cui entrano in gioco dei nuovi progetti. In questo modo, sarà possibile avere un'unica implementazione, corretta e testata, di feature somiglianti tra loro. Come descritto nella sezione **??**, è possibile realizzare soluzioni a prodotto rendendole estendibili (ad esempio, con una libreria), oppure mantenendole all'interno di servizi a sè stanti. Nel caso del progetto in questione, è già successo che si trasferissero logiche all'interno della libreria di base che implementa l'ETL, e si continuerà a procedere in tale maniera. Questa procedura, non deve avvenire solo quando si notano somiglianze con progetti già realizzati, ma deve anticipare il più possibile casi d'utilizzo futuri; nel senso che nel momento in cui si realizzano soluzioni sapendo di poterle riutilizzare in futuro, queste dovrebbero essere implementate "a prodotto" piuttosto che "a progetto".

Un esempio di feature realizzata immediatamente "a prodotto" è quella del reperimento di segreti da *AWS SecretsManager*, descritta nella sezione **??**. Infatti, sono diversi i progetti che necessitano di comunicare con entità esterne, e mantenerne le informazioni di accesso all'interno del codice sorgente rappresenta un rischio elevato di *disclosure*. Di conseguenza, è stato ritenuto opportuno sin da subito parametrizzare la logica di reperimento del segreto dal servizio di *Amazon Web Services* e inserirla in libreria, in modo tale da poterla riutilizzare ed estendere in progetti futuri.

3.4.3 Console per l'aggiornamento di flussi

Uno degli svantaggi dell'architettura individuata è rappresentato dal caso in cui i clienti sbaglino a inviare flussi; in particolare, in casi in cui il flusso non è corretto per una porzione minima, come un singolo campo in un singolo record. La procedura seguita al momento per questi casi consiste nel notificare il cliente dell'avvenuto e attendere il flusso corretto, oppure applicare la modifica manualmente; tuttavia, la seconda opzione non andrebbe seguita in quanto, una volta accordato il tracciato dei flussi, è compito del cliente rispettarlo e

assicurarsi della correttezza dei record inviati. Comunque, tale procedimento può causare dei lunghi tempi di attesa che rallentano e possono bloccare gli sviluppi del progetto; non solo quando si stanno testando feature in ambiente di sviluppo, ma soprattutto quando ne si effettua l'*upgrade* in ambienti *multitenant*. Quindi, la situazione ideale coincide con la possibilità per il cliente di applicare modifiche direttamente sui flussi validati o sui flussi caricati in input sui bucket di *AWS S3*. Questa soluzione si potrebbe raggiungere con la realizzazione di una console *ad hoc* accessibile dal cliente che permetta di visualizzare ed effettuare modifiche ai flussi nelle varie fasi dell'ETL. Tuttavia, per quanto la lettura dei file dai bucket sia relativamente facile, non si può dire lo stesso della loro modifica: infatti, nemmeno le funzionalità di *Athena* consentono di effettuare operazioni di *update* sui file. Però esiste una possibile integrazione con le tabelle in formato *Apache Iceberg* che permette di effettuare operazioni come l'inserimento o l'aggiornamento di record. Inoltre, il design della specifica Iceberg è ottimizzato per il suo utilizzo su Amazon S3, garantendo anche la correttezza dei dati in scenari di scrittura simultanea [**apache-iceberg**]. Riuscendo a realizzare una API per l'aggiornamento di tabelle direttamente sui file di S3, sarebbe possibile utilizzarla in una console in cui il cliente potrebbe effettuare query *on the fly*, eliminando completamente i tempi d'attesa dei casi sopra citati.

Capitolo 4

Descrizione delle implementazioni

In questo capitolo vengono descritte e discusse le implementazioni delle componenti del sistema.

4.1 Manutenzione

Come anticipato nella sezione **??**, il progetto è stato mantenuto su *GitHub*; quindi, il *distributed version control system* utilizzato è stato **Git**. Per cooperare con i colleghi e garantire allineamento sui branch di sviluppo, è stata seguita la convenzione di *GitFlow* con le seguenti caratteristiche:

- Sono stati creati i branch *master* e *develop*, utilizzati rispettivamente per caricare le modifiche sugli ambienti di *staging* (in futuro *produzione*) e *sviluppo*.
- Ogni sviluppatore che volesse applicare delle modifiche, deve creare un **feature branch** a partire da *develop*. Il nome del feature branch deve rispettare il pattern: `feature/<nome-feature>`. Eventualmente, si può includere nel nome del feature branch (prima del nome della feature) anche le iniziali dello sviluppatore che se ne occupa. Si elencano alcuni esempi di feature branch creati in corso di sviluppo: `feature/sr-migrations`, `feature/statistics`, `feature/openSearchWrite`.
- La creazione dei feature branch avviene tramite l'utilizzo del plugin `gitflow` di *Maven*.

- Sono state utilizzate *rebase policy* diverse per quanto riguarda l'allineamento dei feature branch con develop e di develop con master. Nello specifico, per allineare i feature branch con develop è stata utilizzata una policy di **rebase**. Invece, per allineare develop con master, è stata utilizzata la rebase policy di **merge**.

4.2 Implementazioni

In questa sezione vengono descritte le implementazioni realizzate a progetto e a prodotto. In particolare, ci si concentrerà sulle porzioni di codice ritenute importanti, se non essenziali, per la comprensione dell'architettura del sistema a livello implementativo. Le porzioni di codice ritenute troppo verbose vengono riportate nell'appendice **??**.

4.2.1 Implementazioni del progetto

In questa sezione vengono riportati i punti salienti delle implementazioni di ciascun batch del progetto realizzato.

Data Validation

In questa parte viene descritto il batch che costituisce il punto d'ingresso dei dati per l'intero processo di integrazione. Inoltre, siccome è il primo capitolo riguardante l'implememtazione delle componenti del sistema, in questa parte verranno fatte diverse descrizioni e assunzioni valide anche per tutti i seguenti batch.

Il listato **??** mostra come sono organizzati i *main* del progetto. Ognuno rappresenta il punto di ingresso di un batch, per cui è necessario mantenere una determinata struttura. Infatti, il primo comando consiste nel creare una **sessione Spark**, necessaria per poter effettuare operazioni distribuite col framework citato. I comandi successivi rappresentano una serie di chiamate a funzioni di libreria che permettono di caricare in memoria i flussi in base a quanto specificato dalla variabile `jsonContent`. Di fatto, quest'ultima carica un file in formato *json* contenente i flussi da leggere in input con i percorsi indicati nel listato **??** in appendice. Questo genere di caricamento permette di

specificare lo schema di ciascuna tabella in un formato compatto; nel dettaglio, per ogni tabella è necessario specificare la chiave primaria ed eventuali chiavi esterne. In questo modo, nelle fasi successive della validazione sarà possibile controllare se i record sono duplicati o se le chiavi esterne non sono censite.

```
1  object MainData {
2     def main(sysArgs: Array[String]): Unit = {
3        val spark = SparkSession
4          .builder()
5          .appName("data-integration")
6          .getOrCreate()
7
8        spark.sparkContext.setLogLevel("WARN")
9
10       val config = new Config(spark, sysArgs)
11
12       val jsonContent = scala.io.Source
13       .fromResource("conf/ClientMapping.json")
14       .mkString
15       val batchConfig = Config.readBatchConfig(jsonContent)
16       val manager: Manager = config.fromBatchConf(batchConfig,
            Some(CustomDataTransformations))
17
18       manager.parallelWrite()
19    }
20 }
```

Listato 4.1: "Main del batch di validazione dei flussi in input"

Infine, una volta ottenute le configurazioni del caricamento dei flussi, il compito del `Manager` è quello di caricare effettivamente i dataframe in memoria effettuando delle operazioni di trasformazione, per poi riscrivere il risultato in formato *parquet*. Le operazioni di trasformazione vengono effettuate in diversi step, seguendo l'ordine dei controlli necessari per la validazione dei record. In particolare, le trasformazioni vengono definite all'interno dell'oggetto `CustomDataTransformations` che estende l'interfaccia di libreria `Transformations`. Di seguito, si mostra l'interfaccia `Transformations` per meglio spiegare come avviene l'integrazione dei dataframe.

```
trait Transformations {
   def beforeDataValidation(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]

   def afterDataCast(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]

   def afterDuplicatesRemoval(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]

   def afterDiscardedRecordsReuse(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]

   def afterFkConstraintsCheck(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]

   def afterDataValidation(config: Config):
      Map[String, mutable.Map[String, OpTable] => Unit]
}
```

Listato 4.2: Interfaccia con i metodi per realizzare trasformazioni nei diversi step della validazione

Come si può notare dall'interfaccia nel listato **??** l'integrazione dei dataframe presenta diversi step; nello specifico, ogni metodo rappresenta un punto di ingresso per delle trasformazioni dichiarate in maniera funzionale, che vengono eseguite in momenti diversi a seconda del metodo stesso.

1. `beforeDataValidation` - in questo metodo vanno dichiarate tutte le trasformazioni da eseguire prima di qualsiasi step di integrazione. Quando si sceglie di effettuare una trasformazione a questo livello, bisogna tener conto del fatto che mancano diverse informazioni, come il tipo di dato di ogni campo (tutti sono caricati come stringhe).

2. `afterDataCast` - è la fase immediatamente successiva alla trasformazione dei tipi di dato dei campi. Il tipo di dato corretto viene indicato nel mapping, di cui un esempio è indicato nel listato **??** in appendice. I record che presentano incoerenze con i tipi di dato del mapping vengono

scartati (ad esempio, un record con un campo valorizzato con la stringa "*Sì*" con tipo booleano).

3. `afterDuplicatesRemoval` - è la fase immediatamente successiva al controllo delle chiavi primarie. Una chiave primaria duplicata significa un record duplicato. Nel caso di record duplicati, entrambi i record vanno scartati, in quanto non si sa quale dei due sia quello effettivamente correlato alla chiave primaria.

4. `afterDiscardedRecordsReuse` - è la fase immediatamente successiva al riutilizzo di record precedentemente scartati, che sono stati poi corretti.

5. `afterFkConstraintsCheck` - è la fase immediatamente successiva al controllo delle chiavi esterne. Nello specifico, va controllato che una chiave esterna sia censita, ovvero che esista un record nella tabella "*padre*" che presenti dei campi valorizzati come la chiave esterna. Tale procedura è necessaria in quanto successive operazioni potrebbero provocare dei fallimenti, come, ad esempio, delle operazioni di *join* tra tabelle.

6. `afterDataValidation` - in questo metodo vanno dichiarate tutte le trasformazioni da eseguire dopo qualsiasi step di integrazione. Le trasformazioni effettuate a questo livello devono tener conto che le informazioni contenute nei dataframe sono complete e che non verranno effettuati altri controlli in seguito. Di conseguenza, bisogna operare con cautela per non introdurre errori.

L'ultima parte del batch in questione consiste nella scrittura dei dataframe in un unico formato. Nello specifico, ogni flusso in output viene riscritto in formato *parquet*. Tale scelta è dovuta a questioni di *storing* e *performance*. Infatti, i file *parquet* sono binari compressi che occupano meno spazio su storage rispetto ai file di testo. Inoltre, i formati di dati basati su colonne permettono query più veloci rispetto ai file di testo [**parquet**]. In questo modo, i batch successivi che dipendono dalla validazione dei flussi possono accedervi in maniera efficiente da un unico punto.

Scrittura su Database dei Rendimenti

Il batch dei rendimenti utilizza in input i flussi validati dal batch di *data validation* e produce dei file *parquet* con informazioni di sintesi relative ai rendimenti ottenuti dai clienti in un certo periodo di tempo. Quello successivo si occupa di scrivere su un database *Postgres* un sottoinsieme di informazioni relative ai rendimenti calcolati. Inizialmente, queste due operazioni venivano effettuate in sequenza all'interno di un unico batch. Una volta effettuato il deploy e lanciato il job su *AWS Glue* in ambiente di sviluppo, ci si è resi conto che la scrittura su database crea un collo di bottiglia rilevante dal punto di vista delle prestazioni, non risolvibile scalando orizzontalmente con il numero di nodi del job. Nello specifico, il tempo d'esecuzione del processo è più che raddoppiato, passando a oltre 20 minuti rispetto ai 10 precedenti. Di conseguenza, per permettere di eseguire test in ambiente di sviluppo sui rendimenti calcolati separatamente a quelli scritti su database, è stato deciso di separare la logica in due batch.

Nella realizzazione della scrittura su database *Postgres*, è stata implementata una componente utilizzabile generalmente per ottenere segreti da *AWS Secrets Manager*. Le informazioni relative alla connessione al database non possono essere inserite nel codice sorgente per motivi di sicurezza. Quindi vengono mantenute all'interno di un repository di segreti di *AWS Secrets Manager* assieme ad altre informazioni la cui *disclosure* può essere pericolosa. La componente in questione permette di ottenere un qualsiasi segreto definendone la procedura di deserializzazione. Nel caso in esempio, il segreto è salvato in formato *json*, per cui è stato deserializzato usando la libreria *jackson* in una classe.

Il listato **??** mostra come sia stato reso possibile reperire qualsiasi tipo di segreto. La funzione `getSecret` è generica nel tipo di segreto che si vuole ottenere; questo, assieme alla procedura di deserializzazione (resa implicita), può essere definito al livello del caso d'uso. In questo modo, è stato possibile inserire la componente per ottenere i segreti tra i package di libreria interna di Prometeia, affinché possa essere riutilizzata.

```
implicit object PostgresDbSecretProvider extends
    SecretProvider[PostgresDbSecret] {
  override def secret(secretString: String): PostgresDbSecret =
    new ObjectMapper()
        .registerModule(DefaultScalaModule)
        .readValue(secretString, classOf[PostgresDbSecret])
}

case class PostgresDbSecret(
  @JsonProperty("database") database: String,
  @JsonProperty("driver") driver: String,
  @JsonProperty("endpoint") endpoint: String,
  @JsonProperty("engine") engine: String,
  @JsonProperty("password") password: String,
  @JsonProperty("port") port: String,
  @JsonProperty("schema") schema: String,
  @JsonProperty("username") username: String
) extends SecretsManagerSecret

val dbConnectionSecret = sysArgs("--dbConnectionSecret")
val secret = getSecret[PostgresDbSecret](dbConnectionSecret)
```

Listato 4.3: Deserializzazione di un segreto da formato json in una apposita classe `PostgresDbSecret`

Un altro aspetto importante per la scrittura dei rendimenti su database è stata l'implementazione di una query per aggiungere informazioni necessarie e rimuovere le superflue. In particolare, è stata effettuata una operazione di *select* per ridurre il numero di campi da tenere su database e una *filter* per selezionare solo i record di interesse. Inoltre, è stata definita una *User-Defined Function* (più semplicemente *udf*) che permette di applicare una funzione ai valori di una colonna. Quest'ultima è stata utilizzata per inizializzare un campo a seconda del valore di un altro, come mostrato nel listato **??**.

```
private val performanceBucketUdf = udf(netPerformance =>
    getPerformanceBucket(netPerformance))

private def getPerformanceBucket(netPerformance: Double): Int =
    netPerformance match {
  case null => null
  case x if x < -0.1 => 1
  case x if x >= -0.1 && x < -0.05 => 2
  case x if x >= -0.05 && x < 0 => 3
  case x if x >= 0 && x < 0.05 => 4
  case x if x >= 0.05 && x < 0.1 => 5
  case _ => 6
}

var returnItems = spark.read.parquet(RETURN_ITEMS_PATH)
returnItems = returnItems
  .filter(col("level") === lit(1))
  .withColumn("performanceBucket",
     performanceBucketUdf(col("netPerformance")))
  .select("period", "bankCode", "customerCode", "level",
     "netPerformance", "netReturn", "performanceBucket")
```

Listato 4.4: Query per pulire le informazioni da tenere su database *Postgres* con una *User Defined Function*

Statistiche

Il batch delle statistiche ha un output leggermente diverso dagli altri, in quanto deve serializzare delle specifiche informazioni che andranno successivamente interpretate da degli *engine*. Queste informazioni corrispondono a delle richieste convenzionalmente chiamate **massive**, in quanto vengono invocate per ciascun record del flusso dei clienti. Lo scopo del batch è quindi quello di prendere in input tutte le informazioni validate in precedenza e calcolare delle specifiche **statistiche** che andranno poi opportunamente serializzate.

Un aspetto differente rispetto agli altri batch è quello di avere in memoria salvati tutti gli strumenti finanziari. Solitamente, questi vengono consultati effettuando le opportune trasformazioni sull'apposito *Dataset*, quindi semplicemente costruendo il piano logico di Spark senza mantenere alcuna informazione

in memoria locale. In questo batch, invece, è stato necessario riutilizzare delle porzioni di codice *Java* per la costruzione delle richieste. Queste porzioni di codice, oltre a non utilizzare la tecnologia Spark, mantengono in memoria le informazioni relative agli strumenti per diverse verifiche, all'interno di un oggetto chiamato `ProductBO`, ovvero il *business object* degli strumenti. Per cui, è stato necessario deserializzare gli strumenti dai *DataFrame* in una apposita mappa. Per farlo, è stata utilizzata l'**azione `collect`**: questa permette di eseguire il piano logico di Spark e collezionare le informazioni in locale per poterle quindi deserializzare. L'intera procedura è mostrata nel listato **??**.

```
1  // Collect instruments dataframe
2  val mapping = manager.getColumnMapping("stats")
3  val mapper = new ObjectMapper()
4
5  val instruments = manager.loadedTables("strumenti").df
6    .select(mapping:_*)
7    .collect()
8    .map(row => {
9      val instrument = mapper.readValue(row.json, classOf[ProductBO])
10     instrument.getCode -> instrument
11   }).toMap
```

Listato 4.5: Invocazione di un'azione per reperire il contenuto di un *Dataset* in locale.

La funzione `getColumnMapping` preleva da un file di configurazione un insieme di informazioni che permettono di rinominare le colonne di un *Dataset*. Questo viene fatto per avere un'unica notazione in output. Convenzionalmente, viene utilizzato il *camel case*, mentre, solitamente, i flussi ricevuti dal cliente arrivano con le colonne scritte in *upper case*. In questo modo, è possibile definire la trasformazione in un file in formato *json* piuttosto che effettuare una trasformazione per ogni colonna all'interno del codice. Un esempio di *column mapping* è riportato in appendice nel listato **??**.

Successivamente, vengono calcolate delle statistiche in senso generico a partire da informazioni sui clienti; in particolare, per ogni cliente si raggruppano l'insieme dei suoi rapporti con la banca, i saldi relativi a ciascun rapporto e il suo profilo, che riassume i dati sul rischio d'investimento del cliente stesso. Il listato **??** mostra come avvengono le trasformazioni di *group by* e *join*.

```scala
// Group accounts by customer's primary key
val accountsByCustomer = accounts.df
  .groupBy(customers.primaryKey.map(col):_*)
  .agg(collect_list(struct(accountsMapping:_*)).as("rapporti"))

// Group accosalesnts by customer's primary key
val salesByCustomer = sales.df
  .join(accounts.df, accounts.primaryKey)
  .groupBy(customers.primaryKey.map(col):_*)
  .agg(collect_list(struct(salesMapping:_*)).as("saldi"))

// Apply column mapping to riskProfile
val riskProfile = profile.df
  .withColumn("profiloRischio", struct(profileMapping:_*))

// Join customers with grouped tables
val output = customers.df
  .join(accountsByCustomer, customers.primaryKey, "left")
  .join(riskProfile, customers.primaryKey, "left")
  .join(salesByCustomer, customers.primaryKey, "left")
  .select(
    col("rapporti"),
    col("saldiSottostantiByCliente"),
    col("profiloRischio")
  )
```

Listato 4.6: Raggruppamenti di diversi *Dataset* per quello dei clienti e corrispondenti operazioni di *join*.

Si fa notare come, in seguito alla trasformazione di *group by*, sia necessaria un'operazione di aggregazione per riottenere un *Dataset*: tale trasformazione infatti restituisce un `RelationalGroupedDataset`. In questo caso, l'aggregazione corrisponde ad una `collect_list`, la quale restituisce una lista di oggetti relativi ad una espressione-colonna. L'espressione colonna passata come argomento corrisponde al mapping in *camel case* di tutte le colonne dei *Dataset*. Quindi, in questo modo, si vanno a raggruppare tutti i dati relativi a ciascun cliente, senza tralasciare nessun campo.

La fase successiva di questo batch consiste nella creazione delle richieste per gli *engine* a partire dalle statistiche calcolate. Per tale scopo viene sfruttato il

metodo `mapPartitions` che permette di applicare una determinata funzione **ad ogni partizione del *Dataset***.

```
val massives: List[MassiveRequest] = List(
  BreakdownAdeguatezza,
  BreakdownCliente,
  BreakdownPosizione
)

val outputSchema = StructType(Seq(
  StructField("type", StringType),
  StructField("value", StringType)
))
implicit val encoder: ExpressionEncoder[Row] =
    RowEncoder(outputSchema)

stats.mapPartitions(rows => {
  // Create deserialization and serialization object mapper
  val deserializer = new ObjectMapper()
  val xmlSerializer = new XmlMapper()

  // Each row will be mapped into 3 massives
  rows.flatMap(row => {
    val statsInput = deserializer.readValue(row.json,
        classOf[StatisticheInput])

    // Serialize each massive request into xml code
    massives.map(massive =>
      val request = MassiveRequestBuilder.build(massive,
          statsInput)
      val xml = xmlSerializer.writeValueAsString(request)
      Row.fromSeq(Seq(massive.toString, xml))
    )
  })
})
```

Listato 4.7: Invocazione del metodo `mapPartitions`.

Il metodo `mapPartitions`, prende come argomento anche un `Encoder` che specifica la struttura del *Dataset* in output alla funzione di *map*. Trattandosi di un *DataFrame*, l'`Encoder` è generico in `Row` e specifica nomi e tipi delle

colonne. Come si vede nel listato ??, `rows` fa riferimento alle righe contenute in ciascuna partizione del *DataFrame* e, per ognuna, vengono poi create le richieste.

Integrazione Dati *Customer*

L'ultimo batch della sequenza ha il compito di scrivere su un indice *OpenSearch* le informazioni integrate con le chiamate massive agli *engine*. Questo perché poi diversi front end e applicativi *as a Service* andranno a prelevare i dati direttamente dall'indice con delle chiamate REST. Per farlo è necessario ottenere prima tutti i dati sui clienti e integrarli con le risposte ottenute dagli *engine* alle chiamate massive. Una volta fatto ciò, è stato necessario individuare una libreria di AWS aggiornata per l'esecuzione di operazioni *CRUD* sugli indici. Nello specifico, ogniqualvolta si esegua il batch, questo deve effettuare le seguenti operazioni:

1. Creazione di un nuovo indice su cui andranno scritti i nuovi dati integrati e aggiornati.
2. Rimuovere l'*alias* di **master** a tutti gli indici precedenti che lo posseggono.
3. Aggiungere l'*alias* di master al nuovo indice creato.
4. Eliminare tutti gli indici caricati prima di 10 giorni dalla data corrente.

Gli *alias* vengono gestiti in questo modo in quanto le varie applicazioni che fanno uso degli indici, invocano le chiamate facendo riferimento all'*alias* di **master**. In poche parole, si tratta di un meccanismo per avere sempre una versione univoca e aggiornata dei dati sugli indici, riuscendo però a mantenere gli indici dell'ultimo periodo in caso di necessità come eventuali errori o confronti da fare. In questo modo, l'aggiornamento dell'indice master risulta trasparente ai programmi che invocano chiamate su *OpenSearch* e questi non hanno alcuna dipendenza dalla procedura. In particolare, gli indici vengono creati con il seguente pattern per quanto riguarda il nome:

$$< tenant > - < environment > - pfp - < time >$$

dove `tenant` corrisponde al nome del progetto o del cliente a cui si fa riferimento, `environment` corrisponde all'ambiente su cui sta eseguendo il batch

(ad esempio, su ambiente di sviluppo coincide con *dev*) e `time` corrisponde al *timestamp* del momento in cui viene creato l'indice, in modo da poter capire la cronologia degli indici. Invece, l'alias di master corrisponde a una stringa creata con il seguente pattern:

$$< tenant > - < environment > - master$$

in modo da avere una gestione **multi-tenant** degli indici per ogni ambiente di sviluppo.

Per realizzare tutto ciò, è stata implementata una componente che utilizza le librerie di AWS e mette a disposizione dei metodi CRUD per indici e *alias*. Al momento, tale componente è stata inserita nel progetto, ma per sviluppi futuri potrebbe essere integrata in libreria di Prometeia in modo da poter essere riutilizzata da programmi Scala che devono interagire con indici *OpenSearch*. Il listato **??** mostra la procedura di creazione di un indice e conseguente scrittura dei record.

```
1  // Create the index
2  openSearchManager.Indices.create(indexName, classOf[Customer])
3
4  // Write all information about the customers
5  customers.rdd.foreachPartition(rows => {
6    // Connect to the index
7    val localManager = OpenSearchManager(hostname, port)
8
9    // Deserialize customers into chunks
10   val customers = rows.toList.map(row => row.json)
11   val bulks = customers.grouped(bulkSize).toList
12
13   // For each chunk, write on index
14   bulks.foreach(bulk =>
15     localManager.Documents.bulk(indexName, bulk, classOf[Customer])
16   )
17 })
```

Listato 4.8: Creazione di un indice su *OpenSearch* e conseguente scrittura di record.

Si fa notare che, per la scrittura efficiente dei record del *Dataset* contenente i dati integrati sui clienti, viene utilizzata la funzione `foreachPartition`.

Quest'ultima permette di eseguire la procedura specificata al suo interno in ciascuna partizione del *Dataset*. In particolare, si crea una connessione con l'indice all'interno di ogni partizione e si deserializzano le informazioni per ciascun record in formato *json*, cioè il formato con cui vengono mantenute le informazioni sugli indici di *OpenSearch*. In seguito, per garantire una scrittura più efficiente dei record sull'indice, questi vengono divisi in porzioni dette *chunk*. La creazione dell'indice, oltre al nome dello stesso, richiede di specificare una classe specificata *ad hoc*: questo perchè gli indici *OpenSearch* necessitano di un **mapping** che definisca i tipi dei campi al loro interno. Quindi la classe `Customer` contiene il modello dei dati (nomi e tipi dei campi) che appariranno sull'indice. La serializzazione della classe è stata affrontata creando una annotazione apposita, `OpenSearchMapping`, da utilizzare nei campi di `Customer` (parzialmente mostrata nel listato **??**). Durante la procedura, vengono iterati tramite **reflection** tutti i campi della classe che hanno tale annotazione: in questo modo, è stato possibile definire come serializzare ciascun campo con tipo primitivo in un oggetto complesso. Per i campi con tipo non primitivo è stata applicata la stessa procedura in maniera ricorsiva (ad esempio, per costruire una lista di elementi).

```
1  public class Customer extends OpenSearchMapper {
2
3     @OpenSearchMapping(index = false, values = false)
4     public String customerCode;
5
6     @OpenSearchMapping(dataType = "text")
7     public String fullName;
8
9     @OpenSearchMapping(dataType = "nested", parent = true)
10    public List<Balance> balances;
11 }
```

Listato 4.9: Classe contente informazioni sui clienti utilizzata per scrivere su indici *OpenSearch* serializzando i campi con l'annotazione `OpenSearchMapping`.

4.2.2 Implementazioni di Baseline

In questa sezione vengono riportati i punti salienti delle implementazioni di ciascun batch realizzato a prodotto.

Data Validation

L'implementazione di questo progetto si è limitata alla realizzazione del catalogo degli strumenti. Quindi inizialmente vengono caricati solo i flussi necessari per ottenere le informazioni del catalogo degli strumenti finanziari di Baseline. In particolare, sono stati utilizzati flussi *mock*, cioè appositamente creati per sviluppare il programma, ma che comunque rispettano il tracciato di riferimento degli strumenti finanziari: il primo è stato prodotto internamente, mentre il secondo è stato prelevato dal PFP di un progetto. Il primo è stato quindi arricchito di informazioni contenute nel secondo e sono state effettuate delle trasformazioni diverse per output differenti. Nello specifico, per ogni output realizzato è stata effettuata una *left join* tra il flusso prodotto internamente e quello esterno: in questo modo è stato possibile mantenere le dimensioni del primo, arricchendolo con alcuni dati contenuti solo nell'altro.

Successivamente, viene rieseguito il ciclo di validazione e trasformazione dei flussi a partire da quelli prodotti in output dalla parte precedente (che costituiscono il flusso degli strumenti) e da altri contenenti informazioni da inserire nel catalogo. A livello architetturale, questa parte corrisponde a quella effettuata per il progetto. Di conseguenza, si riporta e si commenta nel listato **??** una trasformazione particolare.

```
1  val window = Window.partitionBy("isin")
2    .orderBy(col("value").desc, col("acCode").asc)
3
4  val mainRank = mappatura.df
5    .withColumn("rank", row_number().over(window))
6    .filter(col("rank") === lit(1))
7    .select(col("isin"), col("acCode").as("mainCode"))
8
9  strumenti.df = strumenti.df
10   .join(mainRank, col("code") === col("isin"), "left")
```

Listato 4.10: Applicazione di una *window function*.

Il requisito di questa trasformazione corrisponde ad individuare il record con il campo `value` più elevato per ciascuno strumento (identificato dal codice *isin*) all'interno del flusso `mappatura`. Per farlo, è stata realizzata una *window function*, la quale permette di individuare un rango all'interno di un gruppo di record a seconda di un determinato ordinamento. In questo caso, è stata definita la funzione finestra ordinando i record per il campo desiderato in maniera decrescente; dopodiché, è stata creata la colonna `rank` a partire dal numero di riga successivo all'ordinamento. In questo modo, è stato poi possibile filtrare il *Dataset* per i soli record aventi rango 1, cioè la prima occorrenza di ciascun strumento con il campo `value` più elevato.

4.3 Casi particolari

Questa sezione è dedicata alla descrizione di *feature* particolari, degne di nota, realizzate nei diversi progetti.

4.3.1 Migrazioni

La *feature* per le migrazioni è stata complessa da realizzare in quanto porta con sè una serie di complicazioni. Per migrazioni si intendono cambiamenti di grandi quantità di dati dovuti a eventi specifici o variazioni generiche di informazioni sui clienti finali della banca. Un esempio molto semplice può essere un cliente che cambia banca: questa variazione non provoca la perdita di tutti i dati e le informazioni sul cliente, bensì richiede di applicare una modifica; nel modello trattato, questa informazione è contenuta dalla tabella di anagrafica dei clienti. Nel caso specificato, si tratterebbe di cambiare il campo `CODICEBANCA` ed eventualmente il campo `CODICECLIENTE` nel caso la nuova banca decidesse di assegnarne uno nuovo. La complessità del caso, in realtà, è dovuta al fatto che un cambiamento di un singolo record in questa tabella richiede l'applicazione della modifica in tutti i flussi che presentano una chiave esterna su di essa: ad esempio, la tabella sui rapporti di ciascun cliente con la banca, contiene la *foreign key* della tabella dei clienti; quindi se la chiave del cliente cambia di valore, diventa necessario applicare la variazione anche in quest'altra tabella, in quanto altrimenti si perderebbero le relazioni

tra queste. La situazione diventa maggiormente complessa considerando che alcune migrazioni avvengono su flussi con chiavi molto lunghe, ma soprattutto con molti altri flussi che hanno una dipendenza da essi.

Data la predisposizione dei clienti ad effettuare migrazioni, sia per eventi speciali sia per ordinaria amministrazione, è stato necessario realizzare la *feature* in modo da poterla riutilizzare in occasioni successive. Grazie a *Scala* è stato possibile implementarla in maniera funzionale, come mostra il listato **??**.

```
1  private def applyMigrations(originalTable: OpTable, migrations:
       OpTable, joinCondition: Column, ...): DataFrame = {
2    // Migrations applied to a stream
3    val candidates = originalTable.df.join(migrations.df,
       joinCondition)
4
5    // Out of all the migrated records, keep only those that are
       not already existing in the original table
6    val toMigrate = candidates
7      .join(originalTable.df, originalTable.primaryKey, "leftanti")
8      .drop(migrations.df.columns: _*)
9
10   // Filter the original table keeping the records that DO NOT
       appear in candidates
11   val toKeep = originalTable.df.join(
12     candidates.drop(migrationFields:_*),
13     originalTable.primaryKey,
14     "leftanti"
15   )
16
17   // Union between migrated and kept
18   toKeep.unionByName(toMigrate)
19 }
```

Listato 4.11: Algoritmo generico di applicazione di una migrazione su un singolo flusso.

La *feature* è stata inserita in fase di ETL, nello specifico nello step di *afterDataValidation* spiegato nella sezione **??**: quindi si considera che tutti i dati trattati in questo punto siano validi, in particolare non duplicati e censiti in tutte le tabelle in relazione tramite chiave esterna. L'algoritmo individuato prende in input due flussi:

- il flusso della tabella su cui va applicata la migrazione (`originalTable`);
- il flusso contenente tutte le migrazioni da applicare (`migrations`); queste sono contenute all'interno di un flusso dedicato in quanto possono richiedere decine di coppie di campi, dove una coppia contiene il vecchio valore del campo e quello nuovo.

Sebbene si possa pensare che può bastare una semplice *join* tra questi, in realtà è necessario prendere delle precauzioni. Infatti, c'è la possibilità che un record risultato da una migrazione sia già stato erroneamente inserito dalla banca: in questo caso, l'applicazione della migrazione sul record "vecchio" genererebbe una duplicazione che non verrebbe rimossa visto che tale controllo è già avvenuto. Quindi, la *join* tra i due flussi serve ad individuare dei candidati per la migrazione, ovvero i record da migrare nel caso ideale in cui non ci siano già dati migrati. In seguito, la *leftanti join* rimuove dai record migrati quelli già presenti nella tabella originale. Questo risultato va unito con tutti i dati della tabella originale, esclusi i record in comune con i tutti i candidati precedentemente individuati.

Conclusioni

In questa tesi è stata riportata l'esperienza di tirocinio in Prometeia, cercando di evidenziare il contributo apportato al progetto. Il contesto del tirocinio è stato quello di un progetto in partenza, ma con una *timeline* già ben definita; per cui è stato possibile affrontare i ritmi dell'azienda, imparando già ad adeguarsi alle tempistiche competitive di consegna delle *feature* in progetti importanti. Vista la dimensione e complessità dei progetti in Prometeia, sarebbe impossibile occuparsi dell'implementazione per l'intero progetto, per cui è stata concordata sin dall'inizio una parte che permettesse un importante sviluppo dell'esperienza: infatti, è stato possibile studiare una tecnologia relativamente nuova come Spark in ambiente *cloud* con Amazon Web Services. Quanto appreso ha consentito non solo di affrontare il lavoro da un punto di vista accademico, studiando tecnologie, architetture ed implementazioni in funzione della tesi, ma anche di assumere responsabilità all'interno del team di sviluppo, diventando un punto di riferimento per questioni riguardanti le parti descritte dell'architettura e, in generale, della tecnologia Spark. Inoltre, il tirocinio è stato un ottimo trampolino di lancio nel mondo del lavoro: infatti, il rapporto con Prometeia è continuato dopo il termine del tirocinio.

L'utilizzo di tecnologie all'avanguardia come Spark corrisponde ad un salto di qualità per realtà come quelle di Prometeia: il passaggio al paradigma del *cloud computing* permette un *breakthrough* nelle prestazioni dei servizi forniti, ma anche nell'organizzazione stessa dei team di sviluppo, i quali riescono ad utilizzare risorse in remoto per realizzare software di qualità. L'analisi delle tecnologie riportata in questa tesi, in particolare di Spark e Amazon Web Services, non ha l'obiettivo di effettuare comparazioni con altri *provider* o altre soluzioni, bensì è stata la conferma di un paradigma che Prometeia ha provato in molteplici contesti. L'utilizzo di Spark per la realizzazione di Enterprise

ETL in combinazione con le capacità, in termini di risorse, dei servizi AWS sono risultati efficienti da molteplici punti di vista. Il paradigma Spark non è immediato, ma con una buona conoscenza dei sistemi distribuiti e linguaggi moderni, il passaggio dalla formazione all'essere operativi e autonomi è stato veloce. Inoltre, l'utilizzo di un linguaggio con una sintassi concisa ed elegante come Scala, assieme all'utilizzo di console professionali di AWS, hanno permesso di interpretare facilmente il lavoro svolto da colleghi e di cooperare agilmente con loro, contribuendo in maniera positiva secondo gli standard di Prometeia.

Il tirocinio e il progetto sono iniziati con un approccio accademico, effettuando studi e analisi di problemi e tecnologie, e si sono conclusi con la capacità di lavorare e fare scelte sul campo in completa autonomia. Si reputa quindi che il tirocinio sia stato il perfetto ponte tra l'università e la carriera lavorativa nel campo dell'ingegneria e delle scienze informatiche.

Ringraziamenti

Ringrazio il relatore di questa tesi, il Professor Marco Antonio Boschetti, che mi ha seguito nello svolgimento del tirocinio fino alla stesura della tesi. Il Professore non ha solo fornito supporto in ogni occasione con disponibilità e professionalità, ma ha anche dato consigli preziosi su temi non accademici.

Ringrazio i co-relatori di questa tesi, nonché miei colleghi, il Dottor Luca Nardelli e il Dottor Eduart Mustafa. Entrambi mi hanno seguito nel percorso di tirocinio insegnandomi nuove tecnologie e fornendomi il loro punto di vista professionale per quanto riguarda le diverse sfaccettature della realtà aziendale di Prometeia.

Ringrazio quindi Prometeia e, in particolare, tutti i colleghi con cui ho avuto il piacere di collaborare durante questa esperienza. Gli insegnamenti e l'energia che mi hanno trasmesso sono stati fondamentali per il successo di questa esperienza e per la mia crescita.

Ringrazio la mia famiglia e la Dottoressa Chiara Zammarchi, che mi hanno sempre spronato a dare il massimo e mi sono sempre stati vicini durante il mio percorso di studi. Tornare a casa e trovare qualcuno che mi incoraggia e mi sostiene in quello che faccio è stato l'elemento chiave che mi ha permesso di raggiungere questo traguardo con successo e fierezza.

Ringrazio in particolare il Dottor Alessandro Marcantoni che mi ha aiutato durante i miei studi, sostenendomi con ottimismo e passione. I suoi consigli e la sua vicinanza mi hanno aiutato a prendere le migliori scelte durante questo percorso e hanno reso gli studi un'esperienza indimenticabile.

Ringrazio infine l'Università di Bologna che mi ha permesso di completare questo magnifico percorso e di raggiungere splendidi risultati.

Appendice

Nell'appendice vengono riportati i listati più lunghi, che renderebbero le sezioni precedenti meno leggibili per via della loro estensione.

```
{
 "tables": [
  {
   "name": "anacli",
   "format": "csv",
   "source": "import/CLIENTI",
   "oStorage": "output/store/CLIENTI",
   "iStorage": "import/store/CLIENTI",
   "iErrors": "import/errors/CLIENTI",
   "oErrors": "output/errors/CLIENTI",
   "sep": ";",
   "header": true,
   "primaryKey": ["CODICEBANCA", "C_CLIENTE"],
   "columns": [
    {
     "name": "CODICEBANCA",
     "datatype": "String",
     "nullable": false
    },
    {
     "name": "C_CLIENTE",
     "datatype": "String",
     "nullable": false
    },
    {
     "name": "C_FILIALE",
     "datatype": "String",
     "nullable": true
```

```
      },
      {
        "name": "Z_NOME",
        "datatype": "String",
        "mask": true,
        "nullable": true
      },
      {
        "name": "Z_COGNOME",
        "mask": true,
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "DENOMINAZIONE",
        "mask": true,
        "datatype": "String",
        "nullable": false
      },
      {
        "name": "DATADINASCITA",
        "datatype": "Integer",
        "nullable": true
      },
      {
        "name": "LUOGODINASCITA_COMUNE",
        "mask": true,
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "LUOGODINASCITA_PROVINCIA",
        "mask": true,
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "LUOGODINASCITA_PAESE",
        "mask": true,
        "datatype": "String",
        "nullable": true
```

```
      },
      {
        "name": "C_TIPOSOGGETTO",
        "datatype": "String",
        "nullable": false
      },
      {
        "name": "Z_TIPOSOGGETTO",
        "datatype": "String",
        "nullable": false
      },
      {
        "name": "C_CODICEFISCALE",
        "mask": true,
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "PARTITAIVA",
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "STATO",
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "C_STATO_CIVILE",
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "C_GESTORE_RIF",
        "datatype": "String",
        "nullable": true
      },
      {
        "name": "Z_GESTORE_RIF",
        "datatype": "String",
        "nullable": true
```

```
      },
      {
        "name": "GENERE",
        "datatype": "String",
        "nullable": true
      }
    ]
  },
  {
    "name": "rapporti",
    "format": "csv",
    "source": "import/RAPPORTI",
    "oStorage": "output/store/RAPPORTI",
    "iStorage": "import/store/RAPPORTI",
    "oErrors": "output/errors/RAPPORTI",
    "sep": ";",
    "header": true,
    "primaryKey": ["CODICEBANCA", "C_FILIALE", "C_RAPPORTO",
       "C_SOTTORAPPORTO", "C_TIPO", "C_SOTTOTIPO"],
    "foreignKey": [
      {
        "parent": "anacli",
        "columns": ["CODICEBANCA", "C_CLIENTE"]
      }
    ],
    "columns": [
      {
        "name": "CODICEBANCA",
        "datatype": "String",
        "nullable": false,
        "aliases": ["bankCode"]
      },
      {
        "name": "C_CLIENTE",
        "datatype": "String",
        "nullable": false,
        "aliases": ["customerCode", "holder"]
      },
      {
        "name": "C_FILIALE",
        "datatype": "String",
```

```
        "nullable": false,
        "aliases": ["branchCode"]
      },
      {
        "name": "C_RAPPORTO",
        "datatype": "String",
        "nullable": false,
        "aliases": ["accountCode"]
      },
      {
        "name": "C_SOTTORAPPORTO",
        "datatype": "String",
        "nullable": false,
        "aliases": ["subAccountCode"]
      }
    ],
    "newColumns": [
      {
        "name": "IDENTIFICATORE_RAPPORTO",
        "datatype": "String",
        "nullable": false
      }
    ]
  }
}
```

Listato 12: Mapping che definisce lo schema dei flussi in input e i relativi percorsi di input e output

```
{
  "output": {
    "saldi": [
      {
        "sourceField": "CODICEBANCA",
        "outputField": "codicebanca"
      },
      {
        "sourceField": "C_FILIALE",
        "outputField": "codiceagenzia"
      },
      {
```

```
      "sourceField": "C_RAPPORTO",
      "outputField": "codicerapporto"
    },
    {
      "sourceField": "C_SOTTORAPPORTO",
      "outputField": "sottorapporto"
    },
    {
      "sourceField": "C_TIPO",
      "outputField": "tipo"
    },
    {
      "sourceField": "C_SOTTO_TIPO",
      "outputField": "sottotipo"
    },
    {
      "sourceField": "CODICETITOLO",
      "outputField": "codicetitolo"
    },
    {
      "sourceField": "DT_RIFERIMENTO",
      "outputField": "data"
    },
    {
      "sourceField": "I_VAL_NOMINALE",
      "outputField": "valore"
    },
    {
      "sourceField": "PREZZOAT",
      "outputField": "prezzo"
    },
    {
      "sourceField": "CAMBIO_AT",
      "outputField": "cambio"
    },
    {
      "sourceField": "DT_CAMBIO",
      "outputField": "datacambio"
    },
    {
      "sourceField": "CTVEURO",
```

```
54        "outputField": "ctv"
55      },
56      {
57        "sourceField": "CTVVALUTA",
58        "outputField": "ctvvaluta"
59      },
60      {
61        "sourceField": "DIVISA",
62        "outputField": "divisa"
63      },
64      {
65        "sourceField": "TS_AGGIORNAMENTO",
66        "outputField": "dataAggiornamento
67       }
68    ]
69  }
70 }
```

Listato 13: Mapping che le trasformazioni dei nomi delle colonne di una tabella dalla notazione *upper case* a quella *camel case*.

```yaml
name: Deploy Dev
on:
  workflow_dispatch:
  push:
    branches: [develop]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up JDK 11
        uses: actions/setup-java@v2
        with:
          java-version: '11'
          distribution: 'adopt'
          cache: maven
      - name: Build with Maven
        run: mvn package --settings settings.xml
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.SECRET_ACCESS_KEY }}
          aws-region: eu-south-1
      - name: Copy libraries to s3
        env:
          bucket: s3://basepft-dev-eu-south-1
          path: glue/source/etl
          tenant: basepft
        run: |
          aws s3 cp src/main/scala/it/prometeia/MainData.scala
             ${bucket}/${path}/
          aws s3 cp target/${tenant}-scala-spark-*.*.*-SNAPSHOT.jar
             ${bucket}/${path}/${tenant}-scala-spark.jar
```

Listato 14: *GitHub Action* che gestisce il *deployment* di un programma su ambiente di sviluppo.