UNIVERSITÀ DEGLI STUDI DI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea Triennale in Ingegneria Informatica

Tesi di Laurea

PATTERN ELT PER BIG DATA IN CLOUD

Progettazione di una POC per un istituto finanziario

Laureando
Domenico De Marchis
Matricola 578132

Relatore
Prof. Riccardo Torlone

Anno Accademico 2023/2024

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Se si volesse ricercare la sorgente delle primissime esigenze di ottenere un vantaggio sui propri competitor dall'analisi e l'organizzazione di dati rilevanti per il business, il lontano 1865 è l'albore della Business Intelligence: strumenti e tecniche utilizzate dalle imprese mirate al prendere decisioni informate per il futuro, basandosi proprio sull'analisi ed organizzazione dei dati grezzi per trarne informazioni rilevanti. Incrementare il proprio vantaggio competitivo è sempre stato un obiettivo talmente critico per qualunque azienda che l'avvento di internet e gli sviluppi tecnologici dei primi anni duemila hanno sollevato l'importanza di gestire enormi volumi di dati: i cosiddetti "Big Data". Social network, applicazioni, motori di ricerca... hanno reso esponenzialmente più grande la mole di dati utili alle imprese generata da ciascun utente e, in questa tesi, si sono voluti studiare tutti gli strumenti e le tecniche moderne che permettono, ad oggi, di gestire i Big Data e supportare una Business Intelligence che sia efficiente ed economica. Un'attenzione particolare si è data al ruolo rivestito dalle tecnologie di cloud computing nel Big Data Management, concentrandosi sugli strumenti forniti dal provider Google: la potenza e la scalabilità di Data Warehouse moderni come BigQuery permettono di apprezzare tutti i vantaggi di una transizione da ETL a ELT, esaminando dettagliatamente i processi che permettono di trattare i dati in modo da renderli congeniali all'utilizzo finale, oltre che efficientemente organizzati e reperibili. Questo elaborato di tesi si concentrerà sul mostrare un'applicazione pratica dei concetti e degli strumenti discussi: si analizzeranno esempi tratti da uno dei progetti a cui ho lavorato durante il

periodo di tirocinio, coinvolgente uno dei principali istituti bancari italiani. Per una migliore comprensione, si propone l'analisi un "mockup" di quanto realizzato nel suddetto progetto: l'implementazione di una pipeline di dati tramite software moderni come Airflow permetterà, ancor meglio, di comprendere il ruolo di tutti i concetti e gli strumenti precedentemente discussi nella disamina teorica; Airlfow risulta particolarmente congeniale alle esigenze didattiche, essendo che tutte le funzionalità implementate tramite codice saranno ben visibili a livello grafico.

1

La Rivoluzione Big Data

Nel 2018 l'allora colosso dei social media Facebook, oggi appartenente al collettivo di Meta Platform Incorporated, si trovò a dover affrontare una causa legale su diversi fronti: l'azienda fu imputata di utilizzo improprio e non autorizzato dei dati riguardanti circa 87 milioni di profili iscritti alla piattaforma, violandone le normative di privacy. L'accusa sosteneva che Facebook ne avesse permesso l'accesso alla ormai defunta società di consulenza britannica "Cambridge Analytica", che avrebbe poi utilizzato tali dati per influenzare il comportamento di elettori ed elettrici nelle presidenziali americane del 2016. La suddetta denuncia è uno degli esempi più pertinenti dell'importanza che, ad oggi, rivestono i dati generati da ogni utente: tendenze, comportamenti comuni, previsioni... sono tutte informazioni sensibili, risultato di un'analisi efficiente di grandi volumi di dati, che le aziende utilizzano per valutare il proprio lavoro, collocarsi responsabilmente nel mercato o fornire annunci pubblicitari mirati, come nel caso di Facebook tra le tante. Si pensi all'attività "digitale" di ciascun utente: determinate ricerche su internet, visita di particolari siti web, utilizzo di specifiche applicazioni, acquisto di precisi pro-

dotti... tutti conformi alle proprie preferenze; il concetto microeconomico stesso di "preferenza" è alla base del mercato e dei meccanismi di domanda e offerta che lo guidano. Il concetto di "Big Data", che ci prestiamo di cui a poco ad introdurre, tiene traccia di tutte le preferenze di un consumatore e le "maschera" sottoforma di grandi quantità di dati derivanti dall'attività digitale dell'utente stesso. Le fasi principali del processo di estrazione dell'informazione dai Big Data sono sintetizzate nello schema in figura "1.1", definito "Data Value Chain (DVC)":

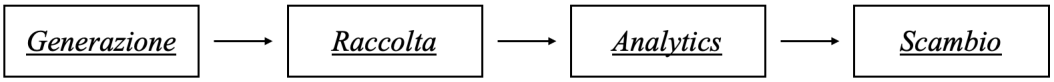

Figura 1.1: Data Value Chain

- Generazione: la prima fase della DVC comporta la generazione effettiva dei dati da varie sorgenti, analogamente a quanto discusso precedentemente con il concetto di "attività digitale"
- Raccolta: fase cruciale dove i dati generati vengono trasmessi dalle loro sorgenti ad un punto comune di archiviazione. I dati passano attraverso un processo di computazione mirato a strutturarli in un formato congeniale alle applicazioni successive di "Analytics", nonché rimuova duplicati, inconsistenze o formati errati, assicurandone la qualità; questa fase comprende tutte quelle operazioni che, generalmente, vengono definite come "data cleansing"
- Analytics: fase dove si realizza concretamente la computazione sui dati e si prendono decisioni conformemente alle informazioni ricavate. All'effettivo la Business Intelligence è circoscritta a questo stadio
- Scambio: l'ultima fase della DVC consiste nell'impiego effettivo delle informazioni estratte in "Analytics"; per "impiego" si intendono principalmente un utilizzo diretto o una vendita a terzi di tali informazioni. Nel caso di un utilizzo interno, questa fase consiste nell'applicazione formale delle decisioni di BI formulate in "Analytics"

In quest'elaborato di tesi ci collochiamo all'interno della fase di "Raccolta" e l'obiettivo che ci proponiamo è uno studio di tutti i principali strumenti, meccanismi e soluzioni moderne di supporto alla Business Intelligence: la fase di "Raccolta" si propone di fornire il dataset su cui verranno effettuate le computazioni ed, in un certo senso, vogliamo approfondire dettagliatamente non tanto la BI ad alto livello, come le tecniche stesse di analisi dei dati e la generazione di report, che comunque citeremo, ma gli aspetti infrastrutturali e tecnologici che sottendono al funzionamento dei sistemi di Business Intelligence moderni, rendendoli efficienti.

1.1 Grandi Volumi di Dati

"Big Data" è un termine la cui primissima apparizione risale all'inizio degli anni '90, sebbene l'exploit che lo ha reso popolare sia riconducibile al 2010. Come la maggioranza dei "neologismi" e le aree di studio moderne, ancora non esistono definizioni e regole a cui è possibile far riferimento per descrivere rigorosamente tutte le caratteristiche e le novità dei Big Data; quello che ci proponiamo di realizzare è una panoramica generale delle differenze tra questi ed altre tipologie di dati, spesso definiti "Small Data" di conseguenza, sottolineando le grandezze con le quali normalmente li si definisce. Il grafico sottostante è uno sguardo alla quantità di dati generati dal 2010 ad oggi, con previsioni per il 2024 e il 2025, misurati in ZettaByte (miliardi di TeraByte):

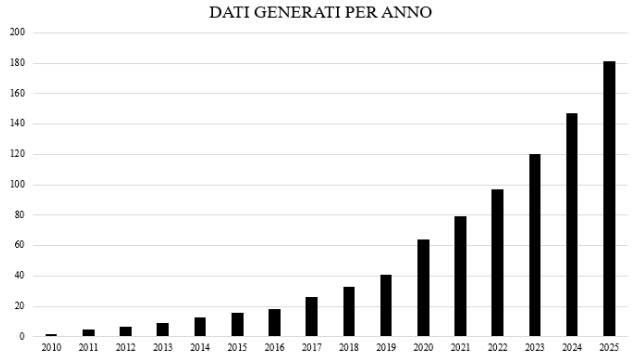

Figura 1.2: Andamento Annuo dei Dati Generati - ZettaByte

L'impennata di dati generati è notevole e dal 2021 procede a ritmi costanti di circa il 23% annuo, che si stima possa essere affidabile anche per 2024 e 2025. Ciò che davvero risalta all'occhio è l'enormità di questi dati: solo nel 2025 si stima che verranno generati 181 miliardi di TeraByte, pari circa alla somma di tutti i dati generati dal 2016 al 2020; se volessimo fare un paragone, per archiviarli tutti servirebbero altrettanti iPhone ultimo modello con massima capacità di storage (1 TeraByte)... Questi risultati sono sintomatici di come le aziende si trovino ad aver accesso ad una mole di dati esponenzialmente maggiore e, valida l'equazione più dati = più informazione, maggiori sono le esigenze computazionali. Se volessimo tenere traccia di tutti gli acquisti effettuati negli store di Roma di una certa multinazionale, allora sistemi di archiviazione dati tradizionali, come i database relazionali SQL, si dimostrerebbero ancora una valida alternativa; tuttavia, le multinazionali hanno migliaia di store nel mondo: se volessimo continuare ad utilizzare il modello relazionale tradizionale, probabilmente già ricavare quanto mediamente un certo cliente spenda, computando miliardi di acquisti, diventerebbero richieste troppo esose... Proprio da quest'esempio si solleva un punto importante: dataset esorbitanti, come quello che tiene traccia delle transazioni nel nostro esempio, si formano comunque sull'unione di dataset più semplici, come quelli di ogni store preso singolarmente. I "Big Data" sono grandi insiemi di "Small Data" che possiamo sintetizzare concettualmente come: dataset la cui dimensione è così vasta da superare le capacità dei sistemi hardware e software normalmente impiegati per gestire ed elaborare dati efficientemente o, comunque, entro tempi ragionevoli. Big Data e Small Data sono ambedue tipologie di dato e, quindi, per nulla differenti a livello logico: indipendentemente dai volumi, entrambe rimangono rappresentazioni intrinseche di informazione e la vera differenza risiede nel come vengano gestiti. I tradizionali database SQL si dimostrano poco efficienti non tanto perché il modello relazionale sia così inapplicabile ai Big Data, ma poiché si necessita di una quantità di risorse, in termini di capacità di calcolo e archiviazione, che non possono soddisfare. Essendo il modello relazionale, a cui sarà dedicata un'intera sezione, lo standard su cui de facto si costruisce la totalità del Data Management, sarebbe auspicabile che questo rimanga

un'alternativa valida anche per Big Data: le tecnologie di Cloud Computing permettono di ovviare a tutte le limitazioni precedentemente discusse, costruendo ambienti dove database relazionali incontrano la scalabilità delle risorse Cloud, potendo stavolta soddisfare la domanda di storage e potenza di calcolo. Il Cloud Computing non costituisce l'unica alternativa possibile, anzi, l'importanza crescente dei Big Data negli ultimi anni ha portato a nuove soluzioni che, abbandonando il modello SQL, si dimostrano altrettano valide a livello prestazionale: framework come Hadoop e database NoSQL sono tra le architetture più utilizzate ad oggi, quali ci proporremo di raccontare in breve nei capitoli successivi. La dimensione dei Big Data, quello che viene comunemente definito "Volume", non è l'unica differenza con gli Small Data: in uno studio del 2001 l'analista Doug Laney ha definito il cosiddetto "modello delle 3V", sottolineando le grandezze fondamentali che sintetizzano la complessità dei Big Data. Oltre che al già discusso "Volume", troviamo "Varietà" e "Velocità", a cui dedichiamo le sezioni successive.

1.1.1 Varietà e Struttura

L'aumento dei dati generati in figura "1.2" sottolinea come, di conseguenza, ciascun utente di internet generi più dati pro capite. Il motore principale di questo fenomeno è riconducibile all'espansione del cosiddetto "Internet of Things (IoT)": termine con cui si fa riferimento alla progressiva estensione di Internet a tutti gli oggetti del quotidiano, la cui digitalizzazione aumenta il volume di dati generati. Smartphones, smart home, assistenti virtuali, smartwatch... nel tempo non cambia solamente l'utilizzo che l'uomo fa della tecnologia, ma soprattutto la precisione della tecnologia stessa: sistemi software sempre più sensibili al catturare dati utili al fornire un'esperienza plasmata dalle attitudini del consumatore stesso, fonte d'informazione fondamentale alle imprese nel comprendere tutte le "preferenze" che muovono il mercato; "Varietà" è una proprietà indicante, appunto, la grande diversità tra le sorgenti che generano Big Data, il che porta a dover gestire dati in formati spesso diversi da quelli "convenzionali".

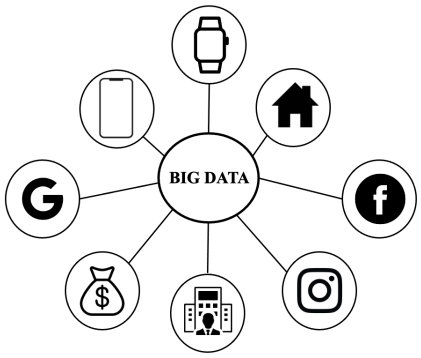

Figura 1.3: Differenti Sorgenti di Big Data

Come discusso precedentemente, lo standard su cui de facto si costruisce il Data Management è il modello relazionale e, come vedremo nella sezione dedicata, questo organizza i dati in strutture ben definite: tabelle di righe e colonne; per il momento si pensi, seppur impropriamente, ad un database relazionale come un grande foglio Excel dove memorizziamo i nostri dati. I database relazionali, duplice la forte rigorosità strutturale, non sanno come comportarsi difronte a dati che non condividono lo stesso formato con il quale sono abituati a lavorare. Se ci venisse chiesto di memorizzare un'immagine in un file Excel probabilmente non sapremo cosa rispondere, analogamente si comporterebbero tutti i sistemi software che automatizzano il processo di memorizzazione dei dati in un relazionale. Questo concetto è così importante che i dati vengono distinti in categorie proprio a seconda del loro formato o "struttura":

- Dati Strutturati: molto spesso si definiscono "strutturati" tutti quei dati che sono organizzati in "strutture predefinite". In realtà trovo ai fini didattici questa definizione un po' fuorviante, per il semplice fatto che, in un certo senso, ciascun dato ha una propria "struttura" intrinseca, indipendentemente da dove questo sarà memorizzato successivamente. Per questo propongo al lettore un'ulteriore definizione che ha una buona sinergia con quello di cui andremo a trattare: si definiscono "strutturati" tutti quei dati che sono facilmente memorizzabili in un database relazionale e computabili da opportuni linguaggi, SQL tra

tutti, o "queryable" come si intende tipicamente in letteratura; esempi semplici di dati strutturati sono fogli di cacolo Excel o risultati di form online, tra i tanti. Anche se impropriamente, un ulteriore esempio di dati strutturati sono i database relazionali essi stessi: i dati contenuti al loro interno sono facilmente trasferibili e gestibili in ulteriori relazionali; questo concetto potrebbe sembrare strano: perché si dovrebbero trasferire dati da un database relazionale ad un altro se questi sono già correttamente contenuti ed organizzati? Quest'ultimo è uno dei concetti fondamentali di questa tesi, difatti tornerà prepotentemente nelle successive sezioni

- Dati non Strutturati: sulla falsariga di quanto detto precedentemente, si definiscono "non strutturati" tutti quei dati che non sono facilmente memorizzabili in database relazionali, tantomeno "queryable" con linguaggi tradizionali. Esempi di dati non strutturati sono: documenti di testo, email, immagini, video, file audio... insomma, la stragrande maggioranza di dati che provengono da applicazioni di largo utilizzo quotidiano, come social network ad esempio. Non risulta così strana la stima effettuata dalla multinazionale di consulenza Gartner Inc. sui cosiddetti "enterprise data": l'$80-90\%$ dei dati che le aziende generano e gestiscono sono non strutturati; la proporzione sul volume generale di Big Data non è così alta, ma comunque i non strutturati rappresentano la maggioranza dei dati prodotti ad oggi e per questo devono essere correttamente gestiti. Con buona approssimazione, è facile capire come il futuro del Data Management sarà, proprio, incentrato nel costruire modelli che possano ricavare più informazione possibile dagli "Unstructured Data" e trattarli con relativa facilità, particolarmente in voga, ad oggi, sono gli algoritmi di Intelligenza Artificiale per l'analisi di immagini, o "NLP - Natural Language Processing" per l'analisi di testo... Sebbene sia vero che i non strutturati costituiscano la maggioranza dei dati in un contesto enterprise, non risulta altrettanto vero che quest'ultimi siano, di fatto, i dati che per le aziende abbiano rilevanza sensibile: in quest'elaborato di tesi parleremo di tutte le moderne

architetture di Data Management, basatesi comunque su dati a forte matrice relazionale, essendo che questo modello si presta ottimamente a descrivere oggetti del mondo reale e di stretta rilevanza per il business; email, video, file audio, immagini... perdono tutte un po' di senso quando si tratta di muoversi sul mercato, o meglio, le informazioni che se ne potrebbero ricavare sono tranquillamente estrapolabili da contesti più agevoli e, per questo, non ne troveremo particolare interesse nelle discussioni a venire

- Dati Semi - Strutturati: generalmente questi sono dati non così semplici da gestire in un relazionale come gli strutturati, tuttavia sicuramente più agevoli dei non strutturati; difatti è una tipologia di dato che sta "a cavallo" tra le due. Esempi di dati semi - strutturati sono file XML e JSON tra tutti, quest'ultimi saranno oggetto di discussione quando parleremo del paradigma NoSQL. Il perché si faccia distinzione tra i dati semi - strutturati e strutturati è principalmente relativo alle esigenze computazionali nella fase "Transform" dei pattern ETL/ELT già accennati precedentemente: qui i dati strutturati, che già sono in un formato congeniale ai database relazionali, passano principalmente attraverso operazioni di "data cleansing" più che trasformazioni di formato effettive, i semi - strutturati presentano una sintassi che rende tali trasformazioni relativamente semplici ma, comunque, necessarie. La maggiore semplicità di processamento dei dati semi - strutturati è data dalla presenza, nel loro formato, di costrutti ricorrenti e ben noti agli algoritmi che li computano, quali "sanno dove e cosa cercare" per la formattazione

1.1.2 Velocità ed Estensione alle "5V"

Nel contesto dei Big Data, "Velocità" si riferisce alla rapidità con cui i dati vengono generati, raccolti e processati. La "Varietà" delle sorgenti che generano Big Data solleva un aspetto interessante e che, finora, ci siamo limitati a sottointendere: con la proliferazione dell'IoT e la generale digitalizzazione,

l'aumento del volume di dati è proporzionale alla sempre più crescente velocità con cui questi vengono generati dalle sorgenti, indicata comunemente come "frequenza di generazione"; si pensi solamente alla mole di dati che generiamo ogni giorno con i nostri dispositivi mobili tramite semplici ricerche su internet o il banale utilizzo di social network... La "sensibilità" dei software moderni al catturare sempre più dati ha come risultato un del tutto naturale aumento della frequenza di generazione. Sebbene non sia tassativamente necessario un processamento "real - time" o "near real-time" di dati in qualsiasi contesto applicativo, è sicuramente vero che questo tipo di bisogni siano sempre più richiesti sul mercato, soprattutto in contesti aziendali: l'analisi di dati in tempo reale permette di identificare trend, cambiamenti, anomalie... nell'esatto momento in cui accadano, ottimizzando i processi decisionali; gran parte delle decisioni di Business Intelligence, infatti, vengono prese proprio analizzando dati in "streaming". Una delle aree di studio più importanti quando si tratta di Big Data sono le reti di calcolatori: i dati viaggiano all'interno di opportune infrastrutture che collegano le sorgenti ai database che li archivieranno e i grandi volumi di questi dati implicano sicuramente delle prestazioni non banali richieste alle reti, quale evoluzione negli ultimi anni, esempio sono i costanti miglioramenti nelle tecnologie di fibra ottica, ha sicuramente contribuito alla capillarità dei Big Data. Uno degli obiettivi di questa tesi è proprio discutere tutti gli strumenti e i meccanismi che permettono di raccogliere e processare i dati tanto velocemente quanto questi vengano generati, discutendo del "percorso" che seguono i dati più che dei mezzi trasmissivi essi stessi. Il "modello delle 3V" fu successivamente esteso al "modello delle 5V" con l'aggiunta di due ulteriori grandezze: "Veridicità" e "Valore". Fin dall'introduzione di questo capitolo si è discusso ampiamente del fondamentale "Valore" che ad oggi rivestono i Big Data e sarebbe, quindi, ridondante discuterne nuovamente; più interessanti sono gli spunti sollevati dalla "Veridicità": più dati, più velocemente e da diverse sorgenti sono la sintesi generalissima che si può realizzare dei Big Data, abbiamo discusso dell'importanza dell'analisi di questi e di tutte le informazioni che se ne possono ricavare e quello che la "Veridicità" sottolinea è la necessità di assicurasi che i dati computati siano accurati, corretti ed affidabili. Gran

parte di quello che permette ai Big Data di essere "veritieri" sono tutti i già discussi processi di "data cleansing" che si occupano non solo di eliminare ridondanze, rimuovere duplicati o verificare che eventuali parametri siano rispettanti, ma anche di provvedere ad eventuali dati mancanti se questi siano ricavabili direttamente. Ai fini di ciò di cui andremo a trattare, al "modello delle 5V" ne aggiungerei una sesta: la "Variabilità". Come discusso precedentemente, l'aumento del volume dei dati generati non accenna a diminuire nei prossimi anni, con stime vicine ai 181 ZettaByte per il 2025... Sebbene queste siano solamente stime e, all'effettivo, il volume potrebbe essere molto di meno, non scordiamo che potrebbe anche essere molto maggiore: le architetture più moderne potrebbero idealmente diventare obsolete nel giro di pochi mesi, all'effettivo magari qualche anno, essendo che questo trend di crescita non accenna a diminuire. La "Variabilità" rende la cosiddetta "obsolescenza" degli strumenti di Big Data Management molto breve, per fare un esempio concreto: se si decidesse di strutturare un certo database per la gestione di Big Data se ne dovrebbe definire la logica di archiviazione dei dati, le performance richieste, la capacità... proprio questi tre concetti sono fondamentali, in quanto tutti determinanti nell'efficienza, o meno, di quest'ultimi. I database sono costituiti da componenti hardware che garantiscono determinate prestazioni in funzione della propria capacità, definita come il volume di dati che si stima questo dovrà sostenere, e qualora questa venisse ecceduta non ci sarebbero più garanzie sulle performance o i tempi di risposta che ci si aspetta prestazionalmente parlando. Costruire database o, più generalmente, sistemi per la gestione di Big Data non basandosi su tecnologie fortemente scalabili perde totalmente di senso, di conseguenza tratteremo dell'applicazione a questo campo della risorsa scalabile per eccellenza: il "Cloud Computing".

1.2 Business Intelligence

In questa tesi discuteremo di tutte le limitazioni portate dai Big Data alle architetture tradizionali e di come queste vengano, storicamente, rese efficienti

ed adattate alle nuove necessità, nonché forniremo una trattazione approfondita su molti degli strumenti più utilizzati nella pratica e li vedremo in opera contestualmente ad uno use case specifico. Tuttavia, ciò di cui discuteremo perderebbe totalmente di senso se, a valle di questi strumenti, non ci fossero tecnologie in grado di interfacciarvisi, sfruttarli ed interpretare i risultati restituiti. Le piattaforme di BI ricoprono un ruolo fondamentale in questo senso, rendendo semplice usufruire della potenza delle architetture Big Data di nuova generazione anche ad operatori non "tecnici", astraendo funzionalità di reporting, generazione di grafici, previsioni e visualizzazione di dati che sono, poi, un must nella gestione d'impresa; Looker (Google), PowerBI (Microsoft) e AWS QuickSight (Amazon) sono solo alcune tra le piattaforme più utilizzate nella pratica, tutte queste permettono di rendere semplici ed immediate operazioni che, invece, pretenderebbero l'iniezione di query SQL anche molto complesse direttamente al database a cui si interfaccino, traducendo particolari input in opportune istruzioni. Tramite l'utilizzo di Dashboard e creazione di KPI (Key Performance Indicator) non soltanto le piattaforme di BI provvedono a tracciare l'andamento esistente del business, ma a rendere più informata la totalità delle azioni di stampo dirigenziale, difatti sono tool con un'importante sinergia con contesti fortemente gerarchici come quelli enterprise; suddette Dashboard, KPI, reportistiche... sono producibili per ciascun livello dell'azienda a diverse granularità, nonché siano, poi, gli strumenti primissimi quando si parli di "Project Management". Architetturalmente, una moderna piattaforma di BI è progettata per essere collegata a Data Lake o Data Warehouse ottimizzati per interrogazioni analitiche, i cosidetti OLAP (Online Analytical Processing) di cui parleremo nel prossimo capitolo, e che costituiscono, in buona sostanza, centralizzazioni per la gestione del dato in contesti aziendali e non. Le piattaforme di BI sono, poi, estremamente soggetti al cambiamento che, negli ultimi anni, il Machine Learning e l'AI stanno portando al mercato IT: particolare enfasi ricoprono i nuovi use case e dashboard di forecasting che implementano, proprio, modelli di Machine Learning built in; proprio in relazione a questo, nella mia esperienza di tirocinio ho assistito ad un progetto riguardante una serie di dashboard Looker progettata per l'analisi e cattura di pirati informatici.

2

Sistemi di archiviazione dati

Come già discusso nel Capitolo 1 - "La Rivoluzione Big Data", gestire efficacemente grandi volumi di dati per trarne più informazione possibile è diventato un elemento sempre più cruciale per il successo di imprese, organizzazioni e individui. Strumento principe per la soddisfazione di tali esigenze sono i database: sistemi organizzativi orientati alla raccolta, archiviazione e gestione di dati, fornendo accesso rapido e affidabile alle informazioni.
Nelle sezioni successive, a prescindere dalla criticità a livello hardware che i Big Data rappresentano per i database nella loro generalità, meglio discusse nella transizione "Cloud" trattata nel prossimo capitolo, discuteremo di come il modello relazionale venga esteso ad un versionamento più moderno, quello "ROLAP", che permetta di gestire dati che, oltre al volume, richiedano di essere analizzati su diverse dimensioni, come quella temporale ad esempio, e di come questo sia sostanzialmente relativo all'utilizzo prettamente pratico del modello relazionale esso stesso, esplorandone diversi use case; è bene tenere a mente che tutti gli accorgimenti ed i best practice che apportateremo al modello relazionale non sono necessariamente sintomatici di un'inadegua-

tezza del modello, anzi, quanto ad una necessità di renderlo più efficiente ed adatto a soddisfare le richieste di analisi di dati in "tempo reale", dove, altresì, verrebbe proprio meno questa caratteristica

2.1 Introduzione al Modello Relazionale

Si definiscono "Relazionali" tutti quei database che organizzano logicamente l'archivazione dei dati in tabelle di opportune righe e colonne. I database relazionali, fin dalla loro primissima definizione nel 1970, trovano ancora ad oggi larghissimo utilizzo proprio per la semplicità nella rappresentazione dei dati: la forma tabellare garantisce, oltre che efficienza nell'organizzazione anche di grandi quantità di dati, una facilità di gestione e comprensione difficilmente comparabile e, proprio per questo, il modello relazionale costituisce, ad oggi, praticamente uno standard a qualunque tipo di architettura di Data Management. Il concetto fondamentale alla base di qualunque database, indipendentemente dal modello relazionale, è la profonda differenza che c'è tra "informazione" e "dato", in quanto quest'ultimi possono significare diverse cose, ovvero fornire diverse "informazioni", a seconda del contesto in cui vengano collocati e i database si pongono, proprio, l'obiettivo di non rendere i dati ambigui; i relazionali sfruttano, come suddetto, opportune strutture tabellari per realizzare ciò. Ciascuna tabella, in un database relazionale, rappresenta una particolare "entità": sostanzialmente, rappresentazioni nel database di "oggetti" del mondo reale. I dati collocativi ne specificano determinate proprietà o caratteristiche, definiti comunemente come "attributi" dell'entità. Il modello relazionale si fonda sull'operazione matematica di "relazione", definita formalmente come il risultato del prodotto cartesiano tra due o più insiemi, quale restituisce, all'effettivo, tutte le possibili combinazioni tra gli elementi degli insiemi presi in considerazione. In un database relazionale si fa particolarmente riferimento al concetto di "tupla" che, spesso e volentieri, viene fatta coincidere ad una singola riga di una delle tabelle del relazionale e, sebbene questo sia macroscopicamente vero, una "tupla" è, in realtà, uno dei risultati restituiti dall'operazione matematica di "relazione"

effettuata tra tutti gli insieme di attributi che definiscono una stessa entità, più semplicemente: una "tupla" è una particolare configurazione dell'entità; una singola tabella di un database relazionale, rappresentante una particolare entità, è costituita da una colonna per ciascun attributo, tramite cui, associando quest'ultime tra loro, il modello costruisce una particolare rappresentazione dell'oggetto stesso, che coincide, appunto, con una singola riga, o "tupla", della tabella stessa. Il concetto di "relazione matematica" impone la restituzione di tutti risultati distinti, ovvero tutte tuple tra loro differenti, da cui ne deduciamo come i relazionali, nella loro generalità, rappresentino entità distinte. Quest'ultimo risultato impone, per costruzione, che in ogni tabella ci sia almeno una colonna con tutti valori differenti: quest'ultima prende il nome di "Chiave Primaria" ed identifica univocamente ciascuna tupla, eventualmente formata da una o più colonne; il concetto di "Chiave Primaria" sarà fondamentale quando, nelle sezioni successive, discuteremo della progettazione concettuale di un base di dati relazionale.
La necessità di effettuare complesse analisi sui dati porta, inevitabilmente, a dover esplorare diverse implementazioni del modello relazionale, principalmente orientate a quali siano le "operazioni più frequenti", ovvero quelle da efficientare: computazionalmente, come vedremo, esistono diversi approcci per migliorare le performance del modello relazionale, come il passaggio da una struttura orientata a righe ad una orientata a colonne, sebbene il contesto Big Data imponga, comunque, di apportare alcuni cambiamenti alla progettazione stessa del modello, che tratteremo di seguito.

2.1.1 Il Linguaggio SQL

SQL, acronimo di "Structured Query Language", è un linguaggio di programmazione progettato per gestire e manipolare database relazionali. Creato nei primi anni '70, SQL fornisce un insieme standardizzato di comandi che consentono agli sviluppatori di interagire con i database ed eseguire molteplici operazioni, tra cui: creazione ed eliminazione di tabelle, aggiornamenti, inserimenti, cancellazioni e modifiche di record, visualizzazioni... nonché varie computazioni; SQL è un linguaggio potente e con una sintassi di facile com-

prensione, punti di forza che si sposano perfettamente con la semplicità dei database relazionali. Il tipo di dialogo che si instaura con un database è spesso incentrato all'ottenere una risposta a determinate domande, tanto che si parla generalmente di "interrogazioni" o "query" al database, per cui SQL fornisce un vasto set di comandi che ci proponiamo di riassumere, fornendone una breve descrizione, in tabella "2.1".

SELECT	Utilizzata per recuperare dati da una o più tabelle, specificando gli attributi (colonne)
FROM	Specifica la tabella o le tabelle da cui recuperare i dati nella clausola SELECT
WHERE	Aggiunge condizioni alla clausola SELECT per filtrare i dati in base a criteri specifici
CREATE TABLE	Permette di creare tabelle specificando attributi e tipizzazione di questi. Definisce anche i nomi significativi con cui si fa riferimento a tabelle e colonne
DROP TABLE	Permette di eliminare una tabella specificandone il particolare nome identificativo assegnato in CREATE TABLE
INSERT INTO	Utilizzata per inserire nuovi record in una tabella. Specifica la tabella di destinazione e i valori da inserire nelle colonne corrispondenti.
UPDATE	Modifica i dati esistenti in una tabella
DELETE	Elimina i dati da una tabella, utilizzata per rimuovere specifiche righe
JOIN	Combina dati da due o più tabelle in base ad una condizione specificata
GROUP BY	Consente l'applicazione di funzioni di aggregazione su particolari gruppi di tuple

Tabella 2.1: Istruzioni Fondamentali SQL

Una famiglia di istruzioni a cui si farà spesso riferimento, in quest'elaborato di tesi, sono le cosiddette "funzioni di aggregazione", quali discutiamo separatamente proprio per sottolinearne l'importanza: si definiscono "funzioni di aggregazione" tutte quelle istruzioni utilizzate per eseguire calcoli su un certo sottoinsieme di dati, restituenti un singolo valore di sintesi risultato di operazioni matematico/statistiche applicatevi. Riassumiamo anch'esse, nelle stesse modalità precedenti, in tabella "2.2". Ciascun database non gestisce autonomamente qualunque richiesta e, in realtà, non è nativamente capace, ovviamente, di comprendere il linguaggio SQL; tutti questi compiti vengono delegati ad uno strato software preliminare: i "Database Management System - DBMS". I DBMS sono sistemi software progettati per la creazione, manipolazione e gestione di database, offrendo all'utente un'interfaccia dove è possibile inserire le proprie query e visualizzare le risposte, nonché molti di questi possano tradurre generici input proprio in istruzioni SQL. Uno dei compiti fondamentali, tra i tanti, dei DBMS e che tratteremo con partico-

lare attenzione in questa sezione è il cosiddetto "memory management": c'è un'importante correlazione tra la logica di memorizzazione e l'efficienza di un database rispetto a determinate operazioni, concetto fondamentale per le discussioni a venire. La breve digressione fornita su SQL è uno dei cardini concettuali di quest'elaborato di tesi: le decisioni di Business Intelligence spesso comportano la necessità di effettuare interrogazioni parecchio complesse, comprensive di molti JOIN e funzioni di aggregazione non indifferenti computazionalmente; come discusso nel "Capitolo 1", tra gli obiettivi fondamentali di questa tesi c'è l'analisi di tutte le soluzioni e gli strumenti moderni per garantire efficienza alle "Business Analytics" e, proprio, l'assicurare tempi di risposta brevi a query computazionali sarà il motivo che ci spingerà, successivamente, a parlare di Data Warehousing.

COUNT()	Restituisce il numero totale di righe soddisfacenti condizioni specifiche
SUM()	Calcola la somma dei valori della colonna indicata, se questa è numerica
AVG()	Calcola la media dei valori della colonna indicata, se questa è numerica
MIN()	Restituisce il minimo tra i valori della colonna indicata
MAX()	Restituisce il massimo tra i valori della colonna indicata

Tabella 2.2: Funzioni di Aggregazione SQL

2.1.2 Progettazione Concettuale di una base di dati

La costruzione di un modello relazionale che rispecchi al meglio la realtà di interesse è un processo complesso ed iterativo, comprensivo di diverse fasi che spaziano dalla progettazione concettuale alla realizzazione logica e fisica tramite la costruzione di schemi basati su una specifica tecnologia, come MySql, Postgres, SqlServer... La progettazione concettuale consiste, in buona sostanza, nel modellare entità e relazioni che, tra loro, catturino i requisiti fondamentali del sistema da realizzare. L'artefatto principe di questo lavoro è il cosiddetto "Diagramma ER", acronimo di "Entity-Relationship", base fondamentale su cui si implementa il database stesso.
La fase iniziale per la costruzione di un buon modello ER e, quindi, di un buon database, è la raccolta dei requisiti da parte degli stakeholder che commis-

sionano o utilizzano il sistema: è una fase molto delicata poiché difficilmente standardizzabile o automatizzabile, nonché espressa in linguaggio naturale, quindi intrinsecamente ambiguo. Utenti diversi usano terminologie diverse, spesso incoerenti e contraddittorie.
Completata la fase preliminare di raccolta, si procede con l'analisi dei requisiti stessi, espressa anch'essa in un linguaggio naturale, ma più tecnico, in quanto effettuata, stavolta, da progettisti e personale di competenza. In questa fase, l'obiettivo è definire un glossario con il giusto grado di astrazione e introdurre uno stile sintattico preciso. Una volta definiti i requisiti, il progettista è pronto per la costruzione di un diagramma Entity Relationship. Il modello ER fu concepito, in letteratura, per fornire una notazione standard alla schematizzazione della struttura logica dei database, indipendentemente dalla tecnologia utilizzata, offrendo il vantaggio di una rappresentazione visiva standardizzata ed intuitiva. Il modello ER si fonda sui tre concetti fondamentali di entità, relazioni tra entità e tra attributi, alcuni già accennati nell'introduzione al modello relazionale, di cui, per una migliore comprensione, proporremo l'analisi di un esempio di diagramma ER, rappresentante una piccola parte di un'applicazione social network in figura "2.1", di cui riportiamo il modello completo e a cui faremo spesso riferimento successivamente; l'implementazione seguente sfrutta la diffusa notazione "Crow's Foot": particolarmente utile, come vedremo, per rappresentare la cardinalità delle relazioni, ovvero il numero di istanze di un'entità che possono essere associate ad un'altra entità.

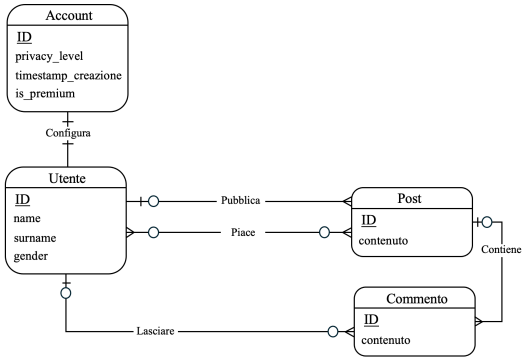

Figura 2.1: Schema ER "Social Network"

2.1.2.1 Entità

Similmente a quanto discusso nella sezione precedente, un'entità è un oggetto astratto o concreto di cui si vogliono catturare determinate caratteristiche e rappresentarle all'interno della base di dati. Tali caratteristiche vengono definite attributi ed hanno dei valori che, congiuntamente, identificano univocamente l'oggetto; in particolare: un sottoinsieme di uno o più attributi è una chiave se identifica univocamente l'istanza dell'entità tra tutte le altre istanze. L'entità può essere considerata una sorta di "blueprint" che definisce come si creino, nel database, le istanze che ne fanno parte, similmente a come una "classe", in un qualunque linguaggio OOP, definisca un "blueprint" per gli oggetti creati tramite la stessa. Visivamente, nel diagramma ER viene schematizzata in un rettangolo bipartito composto da: un'intestazione, riportante il nome dell'entità, e un "payload" che ne contiene tutti gli attributi. Gli attributi si distinguono, poi, in attributi "semplici", come quelli della figura , o "complessi", come ad esempio un attributo "indirizzo", contenente al suo interno via, numero, cap, città... la distinzione che si fa spesso è tra attributi "atomici", ovvero non ulteriormente scomponibili, e "non atomici", quindi ulteriormente scindibili in attributi distinti, concetto che verrà ripreso successivamente.

2.1.2.2 Relazioni

Introducendo il modello relazionale abbiamo gà fornito una definizione, abbastanza superficiale, di relazione, che potremmo formalizzare in: siano D_1, D_2,..., D_n i diversi insiemi di attributi che definiscono una singola entità, allora una relazione R è definibile come l'insieme di n - uple ordinate $\{(v_1, v_2, ..., v_n)|v_1 \in D_1, v_2 \in D_2, ..., v_n \in D_n\}$, dove l'n - upla ordinata $(v_1, v_2, ..., v_n)$ è, in buona sostanza, una singola tupla, ovvero un sottoinsieme del prodotto cartesiano tra tutti i D_i, permettendoci di estendere la definizione di relazione ad una singola tabella del modello relazionale. In un diagramma ER, tuttavia, si esplora il concetto di relazione differentemente da quanto definito in precedenza: si studiano le relazioni come associazioni tra entità. In un ER una relazione lega, quindi, più entità tra loro ed è

principalmente caratterizzata dalla molteplicità con cui possa effettuare tali collegamenti. Si distinguono, in questo senso, le seguenti molteplicità:

- One-to-One: le relazioni "One-to-One" permettono di collegare singolarmente, come suggerisce il nome, ciascuna istanza e_i di un'entità E_i ad una sola instanza e_j di un'entità E_j; questo è il caso della relazione "Configura" tra Utente ed Account in figura "2.1"

- One-to-Many: le relazioni "One-to-Many" permettono di collegare singolarmente ciascuna istanza e_i dell'entità E_i ad un qualsiasi numero di e_j dell'entità E_j; è il caso di un Utente che può pubblicare qualsiasi numero di Post, oppure di un Post che può avere qualsiasi numero di Commenti

- Many-to-One: le relazioni "Many-to-One" permettono di collegare un qualsiasi numero di entità e_i di E_i ad una singola entità e_j di E_j; è il caso di un insieme di entità Commento che, appunto, può appartenere ad un singolo Post

- Many-to-Many: un qualsiasi numero di entità e_i di E_i può essere associato ad un qualsiasi numero di entità e_j di E_j, e viceversa; è il caso di un Utente che può mettere "Like" a diversi Post e diversi Post possono ricevere Like da altrettanti Utenti

ER è uno standard molto più vasto di quello che abbiamo avuto modo di descrivere: estensioni, Entità e relazioni Deboli, ereditarietà... sono solo alcuni degli strumenti per modellare basi di dati reali e molto più complesse dell'esempio, abbastanza didattico, che abbiamo discusso. Nel successivo paragrafo vedremo brevemente come usare la progettazione concettuale come punto di partenza per la creazione un database relazionale.

2.1.3 Riduzione ER in schema relazionali

In questa sezione, mostreremo come tradurre, ad alto livello, il diagramma ER in istruzioni SQL, ovvero come creare lo schema di un database relazionale dal suo modello ER, rispettando i vincoli di Primary Key e molteplicità;

useremo “Postgres”, uno dei più utilizzati RDBMS, per l’implementazione del diagramma ER in figura ”2.1”.

2.1.3.1 Rappresentare Entità

Se E è un entità con attributi semplici $(a_1, a_2, ..., a_n)$, l’entità stessa può tradursi in uno schema relazionale E, una tabella in sostanza, con n attributi distinti.

```
CREATE TABLE Utente {
    ID INT NOT NULL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    gender VARCHAR(5) NOT NULL
};
```

Per attributi complessi si introduce, generalmente, schema relazionale con una foreign key all’interno dell’Entita che punta a questo nuovo schema relazionale.

2.1.3.2 Rappresentare Relazioni

Anche rappresentare una relazione ER comporta la creazione di uno schema relazionale, in particolare: per ogni relazione individuata nel diagramma ER, si introduce uno schema relazionale formato dalle chiavi delle entità coinvolte nella relazione e l’unione tra gli attributi della stessa; questa procedura lascia spazio a ottimizzazioni ed eliminazioni di ridondanza, dipendentemente dalla molteplicità della relazione modellata. Per relazioni “One-to-One” si può evitare di creare tabelle aggiuntive e si può decidere, indifferentemente, se memorizzare la “Foreign Key” in una delle entità coinvolte nella relazione. Volendo fornire un esempio: possiamo scegliere nella relazione “Configura”, tra le entità “Utente” e “Account”, se usare “account_id” nella tabella Utente come una “Foreign key” che punta all’ID della tabella Account, oppure “user_id” con gli stessi meccanismi; modelliamo questa seconda possibilità.

```
CREATE TABLE Account {
    ID INT NOT NULL PRIMARY KEY,
    privacy_level VARCHAR(50) NOT NULL,
    timestamp_creazione VARCHAR(50) NOT NULL,
    is_premium VARCHAR(5) NOT NULL,
    user_id INT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES Utente(ID)
};
```

Per relazioni “One-to-Many” o “Many-to-One” i ragionamenti sono gli stessi, con l’unico accorgimento di aggiungere la “Foreign Key” nello schema relazionale della “parte Many”, non più indifferente come il caso “One-to-One”

```
CREATE TABLE Post {
    ID INT NOT NULL PRIMARY KEY,
    content VARCHAR(255) DEFAULT '',
    creator_id INT NOT NULL,
    FOREING KEY (creator_id) REFERENCES User(ID)
};
```

Per relazioni “Many-to-Many” non è, invece, possibile memorizzare la “Foreign Key” in una delle tabelle coinvolte nella relazione, proprio perchè ogni entità può essere coinvolta più di una volta e cadono i meccanismi precedenti. In questo caso, si crea una tabella intermedia che contiene “Foreign Key” per ogni entità coinvolta nella relazione; ad esempio, per la relazione “Many-to-Many” “Piace”, creiamo la seguente tabella intermedia:

```
CREATE TABLE Piace {
    ID INT NOT NULL PRIMARY KEY,
    creator_id INT NOT NULL,
    post_id INT NOT NULL,
    FOREIGN KEY(creator_id) REFERENCES User(ID),
    FOREIGN KEY(post_id) REFERENCES Post(ID)
};
```

2.1.4 Normalizzazione

Nella sezione precedente si era accennato al concetto di "ridondanza" in merito alla traduzione in schema relazionale del proprio modello ER. Per "ridondanza" si intende, in buona sostanza, la ripetizione di un medesimo dato su più tuple, cui presenza, oltre che un evidente spreco di memoria, causi problemi importanti a tutte le operazioni fondamentali in un relazionale, risultando, quindi, una condizione assolutamente da evitare. Proprio per questo nascono le cosiddette "Forme Normali": particolari metodi di organizzazione della struttura tabellare di un relazionale, utili all'evitare ridondanze e, di conseguenza, tutti quei "problemi" accennati precedentemente, spesso definiti in letteratura "anomalie". Discutendo di tali più in dettaglio:

- Anomalie di cancellazione: presenti quando l'eliminazione di un record non permette di mantenere, almeno parzialmente, alcuni valori all'interno del database, a meno che non si accetti di inserire valori nulli nella chiave associata
- Anomalie di inserimento: presente quando, sebbene si abbiano dati sufficienti, non risulta possibile rappresentare l'informazione se non tramite l'inserimento di valori nulli nelle chiavi associate, spesso sintomatico di uno schema relazionale le cui entità siano, in realtà, scindibili in entità più semplici
- Anomalie di aggiornamento: presente quando l'aggiornamento di un singolo attributo comporti l'aggiornamento di più di una tupla

È dimostrabile, ed evitiamo di farlo poiché esula dagli scopi di questa tesi, che una buona modellazione ER, ed una corretta traduzione in SQL, permetta allo schema relazionale risultato di rispettare le condizioni della "Terza Forma Normale - 3NF", riducendo gli effetti della ridondanza ad una soglia accettabile. Un'aspetto di fondamentale importanza ai discorsi a venire è proprio questo concetto di "Normalizzazione" come organizzazione dello schema SQL in diverse tabelle distinte, aspetto facilmente deducibile dalla traduzione del modello ER in schema relazionale, dove le principali query vengono soddisfatte tramite operazioni di JOIN più o meno complesse a seconda della

molteplicità della relazione; "denormalizzare" vuol dire, in buona sostanza, diminuire il numero di JOIN mediamente eseguiti, concetto che richiameremo spesso nelle sezioni successive

2.2 Database Orientati a Righe o Colonne

Finora ci siamo soffermati ad un livello di astrazione più alto, studiando come vengano organizzati logicamente i dati in un relazionale; quello che vogliamo discutere, adesso, è come vi vengano memorizzati fisicamente. I database relazionali si articolano in due principali tipologie: orientati a colonne e orientati a righe. Questa distinzione nasce proprio dal come si memorizzino i dati: se le righe sono memorizzate in sequenza si parla di "database relazionali orientati a righe", altrimenti "orientati a colonne" se sono quest'ultime ad essere archiviate contiguamente in memoria. Il controllo dei meccanismi di memorizzazione è affidato ai DBMS, già discussi nella precedente sezione su SQL. La distinzione tra database relazionali orientati a righe o colonne risulta di fondamentale importanza poiché, sebbene siano entrambi relazionali, la scelta di uno piuttosto che dell'altro può essere vincolata dal particolare utilizzo che ne si debba fare, o meglio, da quali operazioni siano più frequenti: i database relazionali orientati a righe sono più efficienti quando si tratti di inserimenti, letture, eliminazioni e modifiche, d'altro canto i colonnari sono molto più efficienti quando si tratti di computazioni sui dati, cui operazioni di somma, media, conteggio, calcolo del minimo o del massimo... molto più efficienti, quindi, quando si tratti di fornire risposta a query SQL comprendenti tutte le funzioni di aggregazione già discusse la scorsa sezione. In un database relazionale orientato a righe, la memorizzazione dei dati avviene, come suddetto, contiguamente in sequenza.

Figura 2.2: Memorizzazione Orientata a Righe

Risulta immediato comprendere come uno dei punti di forza di questi data-

base sia proprio la facilità d'inserimento, in quanto l'aggiunta di una nuova tupla comporta unicamente l'accodamento dei dati da inserire, come esemplificato in figura "2.2". D'altra parte, l'eliminazione di un certo record impone una fase preliminare di ricerca: la sequenza di dati memorizzata viene valutata dall'estremo iniziale a quello finale, effettuando confronti per trovare la tupla, o le tuple, da eliminare rispetto ad una determinata condizione. Il vantaggio dei relazionali orientati a righe, anche qui, è l'immediatezza dell'operazione: trovata la corrispondenza, il record è relativamente facile da eliminare, essendo che tutti i suoi attributi sono memorizzati contiguamente in memoria e non sono richieste operazioni ulteriori; si noti come si sia presupposta una logica di ricerca dove venisse valutata la totalità della sequenza di dati, tuttavia si attuano spesso tecniche di hashing o indicizzazione in modo da ottimizzare anche questo processo. Analoghi ragionamenti possono essere effettuati per le operazioni di lettura, rese efficienti dalle medesime conclusioni sulla contiguità dei dati in memoria. D'altro canto, nei database colonnari non sono più le tuple ad essere memorizzate contiguamente, bensì le colonne, ovvero gli attributi dell'entità rappresentata.

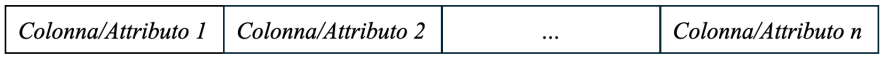

Figura 2.3: Memorizzazione Orientata a Colonne

Spesso, più che una logica a disco singolo, come in figura "2.3", vengono adottate discipline di memorizzazione a dischi multipli: le colonne vengono memorizzate contiguamente in componenti hardware distinte (hard disk, SSD...) e questa soluzione è quella che, all'effettivo, rende i database colonnari l'alternativa più efficiente nel performare interrogazioni comprendenti funzioni di aggregazione, ovvero tutte quelle "query analitiche" strumento principe della Business Intelligence: i DBMS che gestiscono i colonnari conoscono a priori qual è il disco di memoria contenete i dati da aggregare e non hanno bisogno di computare quantità di dati più grandi di quelle che siano strettamente necessarie. Un'esecuzione generica, quindi, di qualunque funzione di aggregazione, valutata su uno o più attributi, viene effettuata solo sui dischi strettamente competenti, a differenza della logica orientata a

righe che, invece, dovrebbe valutare l'intera sequenza in memoria da capo a capo.

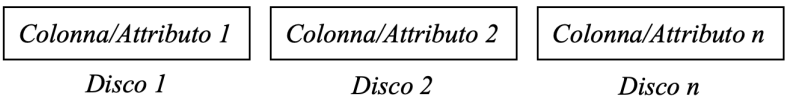

Figura 2.4: Memorizzazione Orientata a Colonne - Dischi Multipli

In database di questo tipo, l'aggiunta di nuovi record comporta l'accodamento dei singoli attributi ai dati preesistenti per ogni disco di riferimento: questo esige, ovviamente, più operazioni di I/O rispetto ai database relazionali orientati a righe, intese come operazioni di lettura e scrittura dei dati. Anche l'aggiornamento di uno o più attributi di una tupla, o di più tuple, implica un numero significativamente più alto di operazioni di I/O rispetto ad una stessa richiesta effettuata ad un database orientato a righe: questo è particolarmente evidente nel caso di aggiornamenti che coinvolgono più di un attributo alla volta, in quanto potrebbe essere necessario operare su diversi dischi invece di effettuare update diretti sfruttando la contiguità dei dati. Ragionamenti analoghi possono essere effettuati per le operazioni di lettura o di eliminazione. Possiamo riassumere tutto in un più semplice: i database colonnari sono molto meno performanti delle controparti orientate a righe quando si tratti di operazioni di inserimento, modifica ed eliminazioni o, più generalmente, operazioni "transazionali", concetto che verrà chiarito successivamente. Questa distinzione è fondamentale ad introdurre le due principali categorie di database più utilizzate, ad oggi, "sul campo" in contesti aziendali: database OLAP e OLTP, ai quali dedichiamo le sezioni a seguire.

2.2.1 Sistemi OLTP

OLTP - OnLine Transactional Processing è una particolare tipologia di database relazionale orientata al supporto di molte delle operazioni con cui si interagisce quotidianamente, all'interno e all'esterno del contesto aziendale: bancomat, online banking, sistemi di prenotazione, gestione degli stock...

Proprio il concetto di "transazione" è fondamentale e, considerando il "Transactional Processing" che costituisce l'acronimo OLTP, talvolta fuorviante. Per "transazione" non si intende l'accezione finanziaria del termine, bensì "transazione di database": insieme di operazioni la quale corretta esecuzione produca un effettivo cambiamento all'interno del database. Una transazione rappresenta, appunto, un'unità logica di lavoro comprendente una o più query SQL orientate principalmente alle operazioni di inserimento, modifica ed eliminazione di record; tali query includeranno le istruzioni di INSERT INTO, UPDATE e DELETE già discusse nella disamina su SQL. In letteratura, tutte le proprietà fondamentali di una "transazione di database" vengono sintetizzate nell'acronimo "ACID - Atomicity, Consistency, Isolation, Durability" che vediamo in dettaglio:

- Atomicity: nei database OLTP ciascuna transazione viene considerata come un processo unico ed indivisibile. I DBMS che gestiscono le transazioni in questo tipo di database fanno particolare attenzione al prevenire aggiornamenti "parziali": l'insieme di operazioni che la singola transazione comprende è tale che o tutte vengono eseguite con successo o il processo viene abortito completamente

- Consistency: la corretta esecuzione di una transazione garantisce il rispetto dei vincoli di integrità del database stesso. Per vincoli di integrità si intende quella classe di proprietà e caratteristiche che tutti i dati nell'archivio devono rispettare, tanto che inserimenti o modifiche devono far si che i cambiamenti effettuati non violino tali vincoli, quali l'inesistenza di valori nulli sulle chiavi, o tuple duplicate

- Isolation: i DBMS OLTP spesso eseguono più transazioni concorrentemente e la proprietà di isolamento garantisce, semplicemente, che lo stato finale del database sia lo stesso che si raggiungerebbe eseguendo le transazioni sequenzialmente

- Durability: i risultati dell'esecuzione della transazione non devono essere più persi nemmeno in caso di malfunzionamenti tecnici. Questa

proprietà spiega come una transazione di database si articoli, poi, nell'aggiornamento anche di eventuali log o storage di backup che riportino tutte le operazioni effettuate sul database per ovviare a queste situazioni di criticità

SQL fornisce delle istruzioni particolari per marcare la query corrente come "transazione di database" al DBMS, riassunte nella seguente tabella:

BEGIN TRANSACTION()	Indica l'inizio della transazione
COMMIT	Formalizza la fine della transazione, unicamente se tutte le operazioni hanno avuto successo
ROLLBACK	Permette di annullare le modifiche effettuate dalla transazione all'occorrenza di errori in esecuzione di una delle operazioni

Tabella 2.3: Istruzioni SQL per Transazioni

Senza indicare la query come una transazione con "BEGIN TRANSACTION" il rispetto delle proprietà "acide" non è garantito: le tre istruzioni sopracitate costituiscono lo scheletro fondamentale per la stragrande maggioranza di richieste che un database OLTP si trova a dover rispondere nel caso più generale. L'esigenza di approfondire al meglio il concetto di "transazione di database" viene naturale se, come nel nostro caso, l'obiettivo è quello di comprendere le principali esigenze progettuali alla base dei database OLTP. Se una transazione comprende unicamente operazioni che alterino all'effettivo il dataset, ciò implica che un database OLTP non si troverà mai a dover rispondere ad una query analitica: l'aggregazione di dati comporta la restituzione di un valore di sintesi che non modifica in alcun modo lo stato corrente del database, a differenza delle suddette operazioni di inserimento, modifica ed eliminazione che, invece, alterano il dataset successivo alla corretta esecuzione di una di queste. Se si decidesse di adottare un modello relazionale, l'implementazione OLTP verrebbe con relazionali orientati a righe proprio perché più efficienti per questo tipo di operazioni, difatti l'applicazione di un database colonnare perde di senso quando si è, come in questo caso, totalmente esenti da qualunque tipo di query analitica. Ulteriore aspetto progettualmente cruciale dei database OLTP è garantire tempi di risposta il più brevi possibile, difatti la restante parte dell'acronimo non ancora discussa

recita “OnLine”: da questo tipo di database ci si aspettano risposte in tempo reale alle richieste degli utenti, nonché una necessità di dati costantemente aggiornati. Un aspetto di fondamentale importanza riguarda la ”profondità storica” dei dati archiviati in un sistema OLTP: in genere ci si concentra sulle transazioni recenti o su un intervallo temporale relativamente breve, più che fornire un dataset accurato e affidabile per finestre temporali più estese. La ragione di questa limitazione risiede nella natura stessa dei database OLTP discussa in precedenza: questi sono ottimizzati per le query transazionali e non analitiche, di conseguenza non ha senso costruire un database capace di gestire ampi volumi di dati su orizzonti temporali estesi poiché la filosofia OLTP si “limiterà” sempre al gestire richieste dove la profondità del dato non è un fattore rilevante. Prendiamo come esempio la massima profondità temporale fornita per lo storico delle transazioni dai sistemi di online banking di quattro tra i maggiori istituti di credito italiani, che riassumiamo in tabella ”2.4”:

Banca	**Storico Transazioni**
Fineco	2 anni
Intesa San Paolo	13 mesi
Unicredit	6 mesi
PosteItaliane	20 movimenti

Tabella 2.4: Tabella degli storici transazioni

Si noti come tutti forniscano una profondità storica delle transazioni limitata e che, mediamente, non superi l’anno e mezzo: i database, ovviamente OLTP, a cui questo tipo di sistemi fanno riferimento forniscono unicamente una vista temporalmente ristretta della totalità delle transazioni effettuate sul proprio conto corrente, ma ciò non vuol dire che non possano ovviare a richieste di questo tipo: rispondono, ma negli intrinseci limiti di ciò che sia rilevante alle richieste che, per la maggiore, gli vengono rivolte e, soprattutto, al cosa vengano progettati per fare. Ovviamente, questo non vuol dire che tutte le transazioni che superino queste soglie siano cancellate, anzi, verranno organizzate in particolari tipologie di archivi dati specializzate

proprio nell'organizzazione e classificazione di grandi volumi di dati dove la profondità storica è un fattore rilevante: i database OLAP, che ci prestiamo ad introdurre. Proprio le differenti profondità temporali dei dati in questione rendono i database OLTP molto meno esigenti a livello di spazio d'archiviazione rispetto alla controparte OLAP.

2.2.2 Sistemi OLAP

OLAP - OnLine Analytical Processing è una particolare tipologia di database orientata all'esecuzione di complesse analisi di dati, finalizzate principalmente al supporto di decisioni strategiche aziendali: utile nel generare report su tendenze di vendite, andamento dei costi, analisi demografica dei clienti, misurazione del successo di campagne marketing... OLAP condivide i principi di "OnLine" e "Processing" di OLTP, che evitiamo di discutere nuovamente, non trattando più però di sistemi "Transactional", bensì "Analitycal": i sistemi OLAP sono "business oriented", ovvero orientati al supporto di analisti, dirigenti, manager... che cercano di trarre informazioni significative dall'indagine su questi dati; proprio il concetto di "Analytical" rende abbastanza palese il collegamento tra quanto discusso nel capitolo sui database relazionali e le controparti OLAP: se si decidesse di adottare il modello relazionale, l'implementazione OLAP comporterebbe una logica colonnare proprio perché estremamente più competitivi degli OLTP nelle computazioni di dati; i DBMS OLAP riescono a garantire tempi di risposta molto migliori a query analitiche, sebbene discuteremo, comunque, di alcuni "meccanismi" utilizzati nella pratica per migliorarne ulteriormente l'aspetto prestazionale. La differenza sostanziale tra OLAP e OLTP sta, quindi, nell'utilizzo prettamente pratico: i database OLTP sono orientati principalmente al supporto di tutti gli "operatori" sul campo, dipendenti di uffici, negozi al dettaglio, sistemi di prenotazione... tutti si interfacciano a questo tipo di sistemi perché l'interesse primario è archiviare i dati e non analizzarli, in maniera diametralmente opposta agli OLAP. Database come quest'ultimi sono molto poco soggetti a operazioni di modifica od eleminazione diretta, tanto che potremmo pensarli come "grandi contenitori" principalmente orientati a raccogliere

dati di natura diversa, dove le uniche "transazioni di database" di rilevanza pratica sono gli inserimenti. Spesso quando si tratta di Business Intelligence e Data Management in contesi aziendali, come nel nostro caso quindi, si fa riferimento ai cosiddetti "Decision Support System (DSS)": i "Sistemi a Supporto delle Decisioni" sono strumenti informatici, prevalentemente software, atti a semplificare il carico di lavoro gravante su tutti i "decision maker", fornendo un'interfaccia che permetta di assecondare le richieste dell'utilizzatore astraendo totalmente quello che succeda a più basso livello, coordinandosi con i DBMS; non è obiettivo di questa tesi discutere l'architettura generale dei DSS, quanto invece sottolineare come la tipologia di database con cui lavorano sia, proprio, OLAP. Discutere delle esigenze che hanno portato alla nascita dei sistemi OLAP risulta fondamentale per una comprensione approfondita di ciò di cui andremo a trattare: l'enorme accumulo di dati negli ultimi anni ha portato alla necessità di realizzare interrogazioni sempre più complesse ai database OLTP tramite query SQL comprensive spesso di molte funzioni di aggregazione ed istruzioni "JOIN", secondo gli stessi meccanismi discussi con la modellazione ER, che ne amplificano enormemente la complessità computazionale. Il bisogno alla base fu quello di costruire dei grandi archivi che potessero permettere non solo di raccogliere ingenti volumi di dati di varia natura, ovviando alle caratteristiche di "Volume" e "Varietà" dei Big Data, ma soprattutto capaci di organizzarli e strutturarli in modo tale da rendere efficiente la loro analisi e ridurre i tempi di risposta delle query che gli vengono rivolte, rimanendo coerenti al principio "OnLine": lo schema a stella, la rappresentazione multidimensionale dei dati e il Data Warehousing costituiscono tre tra le principali rivoluzioni del Data Management moderno che ci proponiamo di raccontare. OLAP può essere inteso più come la filosofia con cui si trattino le query di business analytics rispetto al come vengano gestite all'effettivo, difatti tanto di questo dipende dalla particolare logica di gestione dei dati che si adotta e noi ci siamo, in un certo senso, vincolati al modello relazionale poiché, oltre ad essere semplice e quello ad oggi più utilizzato, faremo affidamento al relazionale "Google BigQuery" nella parte progettuale conclusiva; le due declinazioni fondamentali di OLAP che hanno trovato, nella pratica, campo fertile sono ROLAP e MOLAP, quali andremo

ad approfondire in dettaglio.

2.3 Data Warehousing

Per spiegare il concetto di "Data Warehouse (DW)" ci avvaliamo della definizione data da William H. Inmon, padre concettuale delle DW: "Un Data Warehouse è una raccolta dati integrata, orientata al soggetto, variabile nel tempo e non volatile, di supporto ai processi decisionali"... Anche a costo di risultare banali, probabilmente non esiste altra maniera di comprendere al meglio le caratteristiche fondamentali di un DW; ciononostante, questa descrizione, forse, non è sufficiente per cogliere il concetto alla base: un Data Warehouse non è nulla di così distante da un archivio dati tradizionale, difatti molte delle Warehouses utilizzate ad oggi si fondano, comunque, sul modello classico di database relazionale, concentrandosi sull'integrare dati da diverse fonti e organizzare la totalità dell'informazione in maniera ottimale; le Data Warehouse sono, infatti, database OLAP e continuano a valere tutte le proprietà precedentemente discusse. Un'aspetto di fondamentale importanza, che ci limitiamo solo ora a discutere, è l'analogo OLAP della "profondità storica" dei dati archiviati: se i sistemi OLTP gestiscono dati temporalmente limitati, gli OLAP, e le DW di conseguenza, organizzano l'informazione su orizzonti molto più ampi, rendendoli l'archivio dati principe delle business analytics, come avevamo già accennato d'altronde, permettendo di ottenere una visione completa e approfondita dell'evoluzione dei dati nel tempo, identificare trend e cambiamenti quali sono, spesso, tra le esigenze più comuni. Per capire il 'come' avvenga questo, immaginiamoci un Data Warehouse simile ad un archivio che si rinnovi ciclicamente: i dati sono aggiornati ad intervalli regolari tramite la realizzazione di una sorta di "istantanea" dei dati operazionali a quello specifico momento e, organizzando le nuove informazioni con le preesistenti, si costruisce un database che cresce continuamente nel tempo. Così possiamo corroborare quanto detto precedentemente: il costante aggiornamento delle informazioni in un DW è esente da qualunque tipo di limitazione temporale, anzi, si predilige un'orga-

nizzazione efficiente dell'evoluzione storica dei dati piuttosto che concentrarsi solo su un orizzonte temporale limitato, per esempio quello delle transazioni più recenti; le esigenze progettuali, banalmente a livello di storage capacity, sono ben diverse nei DW e gli OLAP rispetto agli OLTP. Data Warehouse e database OLTP rispondono a query con 'scope' temporali molto differenti e ciò comporta, inevitabilmente, anche una differenza sostanziale nell'utilizzo prettamente pratico di questi: le DW sono database in sola lettura e le query a cui rispondono non comprendono inserimenti, eliminazioni o modifiche, bensì ottimizzano la visualizzazione di dati e l'efficienza di operazioni complesse; le informazioni vengono raccolte, come accennato nella sezione precedente, dai sistemi OLTP "sul campo" e da questi, ciclicamente, le DW prendono informazioni, ma non si contemplano operazioni di inserimento diretto, da intendersi come query SQL della famiglia "INSERT INTO...".
Un Data Warehouse è un archivio dove confluiscono tutti i dati operativi da ciascun database OLTP: a livello pratico, per un impresa potrebbe rappresentare una sorta di "grande magazzino", "warehouse" appunto, dove tutti i dati sensibili, come quelli di Marketing, Economici o quelli relativi alla privacy, sicurezza, soddisfazione del consumatore... siano facilmente reperibili, indipendentemente dalla profondità storica del dato. A questo punto, la definizione di Inmon risulta immediata:

- Le DW sono "orientate ai soggetti di interesse" poiché ruotano attorno ai concetti d'interesse dell'azienda: clienti, prodotti, vendite, dati finanziari, pubblicità... organizzandoli
- La "variabilità nel tempo" è data dal rinnovamento ciclico delle informazioni nelle Warehouse
- Le DW sono "non volatili" poiché archivi immutabili in sola lettura e non modificabili direttamente
- Le DW sono "di supporto alle decisioni" per gli stessi motivi dei database OLAP

Si noti come OLAP e Data Warehouse siano stati spesso e volentieri discussi congiuntamente. La domanda che potrebbe sorgere al lettore è perché, allo-

ra, si faccia una distinzione tra questi due database se, alla fine, condividono moltissime proprietà e l'aspetto applicativo sembra essere lo stesso... Questa percezione deriva dal fatto che le Data Warehouse sono profondamente OLAP e ne condividono la totalità delle caratteristiche, tuttavia OLAP è più una filosofia, come già accennato precedentemente, che un'approccio effettivo alla progettazione di database: i Data Warehouse prendono tutti i principi OLAP e li mettono in pratica. Quello che davvero i Data Warehouse aggiungono agli OLAP è l'integrazione tra i dati, concetto fondamentale soprattutto in ambito enterprise: i database OLAP si concentrano sull'ottimizzare le performance analitiche sui dati che dispongono, i Data Warehouse garantiscono che quest'ultime siano effettuate sui dati più recenti, costantemente aggiornati e contenuti in un database capace di integrare ed unificare i dati provenienti sia da fonti interne che esterne all'impresa, costituendo un'ambiente centralizzato dove siano contenuti tutti i dati sensibili nel loro insieme e dove si eviti di far riferimento a database distinti. L'integrazione che portano le Warehouse, appunto, risolve tutti quei problemi di comunicazione definiti come "data silos": gestione dell'informazione organizzata in più database distinti e non in grado di comunicare tra loro, aspetto che rende difficile computare e visualizzare i dati nel loro insieme. I "silos" sono molto comuni nei contesti dove sono presenti forti strutture gerarchiche, cui le aziende sono esempio principe: ciascun dipartimento, filiale, sede... sono considerabili "silos" se non correttamente integrati tra loro; il cuore del discorso è proprio questo: il Data Warehousing non disicentiva l'utilizzo di database distinti per una corretta archiviazione di tutte i dati che vengono utilizzati e generati dai singoli dipartimenti, filiali, sedi... quanto sottolinea l'importanza che tutti questi comunichino tra loro passando attraverso un un unico grande tramite, che è proprio la Warehouse essa stessa.
In un certo senso, le DW sono concettualmente molto simili ai vecchi "Mainframe" e, come discuteremo successivamente, le architetture più moderne potrebbero essere intese come evoluzioni di quest'ultimi. Tutto ciò che abbiamo discusso non è, ovviamente, vincolato alle sole fonti interne all'azienda, anzi, le Warehouse devono esser capaci di poter far confluire in sé dati anche da fonti esterne all'azienda, se quest'ultime sono sensibili al business, per gli

stessi concetti di integrazione precedentemente discussi. Leggendo un po' tra le righe, è chiaro come i Data Warehouse si pongano l'obiettivo di risolvere tutti quei problemi che proprio i Big Data portino alle strutture di archiviazione dati tradizionali: potenti database progettati per lavorare con grandi volumi di dati e gestire l'alta velocità con cui vengano generati, integrando la varietà delle sorgenti in un unico ambiente. Quello su cui vogliamo concentrarci maggiormente è la "Varietà": come abbiamo visto nel "Capitolo 1", quest'ultima non fa riferimento solo alla diversità tra le sorgenti, ma anche e soprattutto ai diversi "formati" che questi possono assumere, distinguendoli in dati strutturati e non strutturati; i Data Warehouse, dovendo fornire alte prestazioni computazionali e risposte in tempo reale, tollerano solamente dati strutturati nel formato con cui sono abituate a lavorare, ottimizzate quindi per una determinata e specifica logica di gestione del dato. Tuttavia, nei moderni contesti enterprise i dati non strutturati rivestono un'importanza sempre più cruciale e, a questo proposito, le architetture moderne di Data Warehousing si basano su concetti più "totalizzanti" rispetto a ciò che abbiamo finora discusso: sebbene gli unstructured data non siano di stretta rilevanza computazionale, far sì che le DW possano archiviarli e supportare sistemi che li gestiscano risolverebbe, analogamente, il concetto stesso di "Data Silos" anche per gli unstructured data; tutte le moderne implementazioni di Data Warehousing supportano, infatti, un particolare layer di storage che, come vedremo, non faccia alcuna distinzione sul particolare formato dei dati, sebbene computi solamente i relazionali. Un altro tra gli aspetti di più rilevante importanza di come i moderni Data Warehouse si discostino dal modello tradizionale, riguarda, proprio, le transazioni "ACID": sebbene i DW non nascano per supportare operazioni di questo tipo, i concetti di "disaggregazione tra computazione e storage" portati dal "Cloud Computing" o, più generalmente, la grande capacità computazionale delle architetture moderne, permette di poter sostenere anche operazioni di questo tipo, rese terribilmente inefficienti in quanto, all'effettivo, l'obiettivo dei DW è diametralmente opposto: performare ottimamente su operazioni computazionali e non transazionali. Introdurre e discutere dei Data Warehouse, oltre ad essere uno strumento standard nella stragrande maggioranza delle architet-

ture moderne, permette di concretizzare tutte le criticità che portano i Big Data alle architetture tradizionali: le DW sono tra i database principi dei sistemi informativi enterprise, come alla base dei sistemi SCADA nelle reti industriali ad esempio, entrambi sono esemplificazioni di contesti in cui il progresso tecnologico ha portato ad un aumento esponenziale del volume di dati prodotto e da gestire, sottolineando tutte le inefficienze di modelli intrinsecamente non adatti. L'evoluzione, nel tempo, delle architetture di Data Warehousing coincide, poi, con la storia stessa delle diverse soluzione che si sono alternate negli anni, fino a convergere nelle correnti architetture di "Cloud Data Warehouse", che discuteremo successivamente.

2.3.1 MOLAP

Come accennato precedentemente, il Data Warehousing si articola in due principali filosofie di gestione dati: ROLAP e MOLAP; in questa sezione ci proponiamo di approfondire quest'ultima, che sarà utile successivamente per parlare proprio di ROLAP. Multidimensional OLAP o, più semplicemente, MOLAP è un particolare modello basato sull'utilizzo di strutture multidimensionali per la gestione di dati, abbandonando quindi la logica relazionale. MOLAP si basa su tre concetti fondamentali:

- Fatto: evento misurabile e soggetto principale dell'analisi che si vuole effettuare, coerente al particolare contesto in cui si colloca il DW
- Misura: determinata proprietà caratterizzante il Fatto e di interesse all'analisi, tipicamente "atomica", non ulteriormente scomponibile
- Dimensione: particolare orizzonte lungo il quale si vuole effettuare l'analisi in questione

Quest'ultimi non risultano così eccessivamente differenti dai concetti di entità ed attributo del modello relazionale: in un certo senso, potremmo intendere un Fatto come un'entità che si voglia analizzare, le Misure come i suoi attributi e le Dimensioni come un particolare attributo che "sviluppa" tutte le Misure lungo una certa prospettiva. Anche qui, a costo di risultare

banali sfrutteremo uno degli esempi più utilizzati e potenti in letteratura: le “Vendite” di un generico prodotto. Volessimo organizzare quest'ultime multidimensionalmente, potremmo considerare:

- Il “Fatto” come la “Vendita” essa stessa, essendo quest'ultima proprio il soggetto dell'analisi
- “Quantità” e “Incassi” come misure caratterizzanti l'analisi di vendite nella stragrande maggioranza dei casi
- “Tempo”, “Prodotto” e “Negozio” come dimensioni, essendo tutti questi attributi che contestualizzano e specificano l'analisi delle vendite sotto una particolare prospettiva

Avendo scelto 3 dimensioni nel nostro esempio, possiamo pensare ad una modellazione multidimensionale come quella in figura ”2.5”.

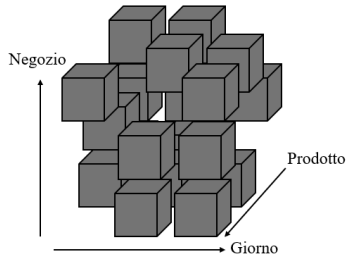

Figura 2.5: Cubo MOLAP - Vendita

Proprio il concetto stesso di dimensione ha dato vita al modello cubico, dove, ovviamente, non si è vincolati ad un massimo di 3 dimensioni, anzi, idealmente si potrebbe far riferimento a modelli n - dimensionali o, più propriamente, ipercubici; si noti come questo tipo di rappresentazione sia tanto di facile comprensione quanto quella relazionale, rappresentandone una valida alternativa sotto certe condizioni. Se il modello relazionale lavora per “tuple”, il modello multidimensionale memorizza i dati in “celle”: qui facciamo della figura 2.5 virtù, nel senso che per “cella” si intende proprio ogni singolo cubo che costituisce la struttura nel suo insieme e che, al suo interno, conterrà i dati all'effettivo. Sebbene possa non sembrare proprio una definizione rigorosa,

rende ottimamente l'idea. Proprio le "celle" ci permettono di rimarcare un concetto fondamentale: i modelli multidimensionali, come quelli relazionali, sfruttano ipercubi e tabelle puramente come logica di visualizzazione/rappresentazione e, a livello fisico, ovviamente non dobbiamo pensare che la memorizzazione dei dati avvenga in questa maniera; nel modello multidimensionale si parla di "celle" proprio perché l'implementazione più diffusa comporta l'utilizzo di array bidimensionali, cui in letteratura difatti ci si riferisce spesso a "celle di array". Continuando con quest'approccio di analogia al modello relazionale, le "tuple" definiscono determinate configurazioni dell'entità rappresentata: una determinata vendita di un certo prodotto, in un certo negozio, in una specifica data, con uno specifico incasso, in una particolare quantità... se si pensasse "relazionalmente", le tabelle rappresenterebbero tutte le differenti transazioni effettuate per la vendita di un certo prodotto e le tuple, quindi, sarebbero molto vicine concettualmente alle singole voci di uno "scontrino"; con il modello relazionale, le tuple coincidono spesso e volentieri con eventi reali e del contesto strettamente applicativo del problema. Quest'aspetto non è, invece, garantito nel mondo multidimensionale: i dati contenuti all'interno di ogni "cella" sono pre - aggregati, ovvero si passa per un iniziale stato di pre - processamento dove vengono effettuate varie aggregazioni sulle Misure a seconda del caso, principalmente somme, aspetto che già ci fa intuire come un Fatto non coincida tassativamente con un evento reale, difatti le singole "celle" rappresentano quantità ed incasso complessivi di un certo prodotto, venduto ad una determinata data, in uno specifico negozio, risultato dell'aggregazione di tutte le vendite singolarmente organizzate nel modello relazionale. In figura "2.5" le singole celle rappresentano le vendite giornaliere complessive di un negozio divise per prodotto, si noti come alcune "celle" risultino mancanti poiché, chiaramente, non si vendono tutti i prodotti tutti i giorni in tutti i negozi... In questo particolare esempio, si è organizzata la dimensione Tempo per "Giorno": ovviamente, essendo i modelli MOLAP sempre orientati al semplificare la visualizzazione e l'analisi di dati, in questo modo non solo l'informazione, ma anche la struttura stessa di cubo risulta poco chiara e di difficile interpertazione, indipendentemente dalle aggregazioni effettuate per ogni cella. Per ovviare a questo problema, i

linguaggi di interrogazione per questa logica di gestione dati, tra cui MultiDimensional eXpressions - MDX ad esempio, forniscono una serie di operazioni che si basano sul concetto di "livello di aggregazione" delle dimensioni: ciascuna di queste è organizzata in diversi livelli di "granularità", raggiungibili a seconda delle particolari aggregazioni tra celle che è possibile effettuare; infatti sommando le vendite giornaliere di un certo prodotto per ogni giorno dell'anno otterremo, ovviamente, le vendite annuali per tale negozio... e questo è, in soldoni, il meccanismo di pre - processamento con cui vengono inizialmente creati gli ipercubi. Una possibile organizzazione dei livelli potrebbe essere come quella in figura "2.6".

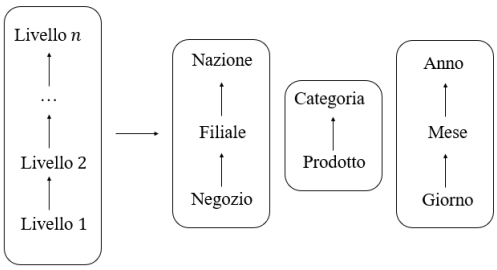

Figura 2.6: Granularità Dimensioni

L'aggregazione da Livello 1 a Livello n è bidirezionale: posso decidere sia di aumentare il livello di dettaglio, scendendo nella gerarchia dei livelli, come di diminuirlo, risalendo; ovviamente i Livelli 1 dipendono da come i dati vengono raccolti dai database operazionali sul campo, ovvero: se venisse registrato il mese e non il giorno di una singola vendita non posso sperare di scendere ulteriormente nella gerarchia dei livelli e risalire proprio al giorno della singola vendita, poiché non direttamente disponibile. Le operazioni che, finora, ci siamo limitati unicamente a sottointendere si definiscono "roll - up" e "drill - down": entrambe cambiano la granularità delle dimensioni di riferimento, riorganizzando il cubo a seconda del particolare livello di dettaglio; a queste si aggiungono le operazioni di "slice and dice" che, sostanzialmente, consistono in analoghi del SELECT SQL, costituendo un parco di operazioni fondamentali alla base di qualunque gestione multidimensionale dei dati. Sebbene risulti chiaro come uno dei principali vantaggi dell'approccio MOLAP sia una

maggiore efficienza e flessibilità nella gestione delle query analitiche, proprio la pre - aggregazione si dimostra, in questo senso, un'arma a doppio taglio: MOLAP è una filosofia che non permette di performare ottimamente per grandi volumi di dati poiché quest'ultimi vengono, proprio, pre - aggregati. Il modello MOLAP tradizionale proponeva che tutte le aggregazioni, per ogni possibile combinazione di dimensioni e livelli di granularità, venissero effettuate nella fase iniziale di creazione del cubo, indipendentemente da quali fossero, poi, le query a cui dovrebbe rispondere: questo approccio, "brute - force" che si voglia, permette di ottimizzare l'aspetto prestazionale di questi database minimizzando i tempi di risposta, risultando, però, insostenibile non solo a livello computazionale, ma anche di storage capacity per grandi volumi di dati; si pensi, ad esempio, al concetto di "celle mancanti" in figura "2.5", per l'implementazione di accesso posizionale tramite array ai dati una "cella mancante" impone che, comunque, si sia allocata memoria per tale posizione anche se, all'effettivo, non si utilizzi... MOLAP fu, infatti, una piccola rivoluzione nei primi anni 2000 e l'avvento dei Big Data non ha sicuramente contribuito alla sua estensione, anzi, il futuro di MOLAP, nonostante tutto il lavoro effettuato su soluzioni di aggregazioni parziali, risiede in soluzioni ibride con il modello ROLAP, cosiddetti sistemi "HOLAP - Hybrid OLAP", che non tratteremo poiché poco pertinenti ai discorsi a venire. Proprio il modello ROLAP si dimostra, ad oggi, di più largo utilizzo poiché, sebbene sia meno efficiente, riesce a garantire prestazioni ottimali a parità di risorse di calcolo per dataset molto molto più estesi.

2.3.2 ROLAP

Come ampiamente trattato nella sezione precedente, il modello MOLAP spicca come una soluzione progettuale "ad hoc" per l'organizzazione di dati in un DW: quando l'analisi dei dati ha l'esigenza di concentrarsi su determinati orizzonti, come quello temporale ad esempio, fa di questa necessità il cuore concettuale del modello, dimostrandosi efficiente nelle computazioni su volumi di dati contenuti. La necessità di estendere la possibilità di effettuare analisi dimensionali anche al modello relazionale viene, sostanzialmente, ol-

tre che dalla grande diffusione di questo modello, da tutto il lavoro che è stato svolto sui database relazionali e l'esperienza che si ha nel trattare con quest'ultimi, soprattutto nei contesti enterprise. Non essendo i concetti di "Misura", "Dimensione" e "Fatto" nativi del modello relazionale, si cerca di trasportarli analogamente a quanto detto nella sezione precedente: i Fatti possono essere rappresentati tramite singole tabelle (relazioni), le Misure come attributi di queste e per le Dimensioni si sceglie, nonostante anche qui si sia concettualmente molto vicini agli attributi, di organizzarle in tabelle distinte e direttamente collegate al Fatto in questione. Tutte le considerazioni che si fanno in ROLAP sono orientate a risolvere il principale problema che il modello relazionale tradizionale avrebbe se fosse "dato in pasto" ad un qualunque contesto soggetto alla complessità dei Big Data: Data Warehousing vuol dire, nell'accezione più generale del nostro caso di studio, implementare database che computino efficientemente grandi volumi di dati, per cui il problema principale dei relazionali che lavorano su dataset estesi è dato dalla complessità delle numerose istruzioni di JOIN quali i DBMS si troverebbero a dover gestire; indipendentemente da tutte le tecniche, cosiddette di "indexing", che si possano adottare per diminuire la complessità computazionale di quest'ultime, i volumi spropositati dei Big Data fanno sì che, mediamente, la complessità dei JOIN sia quantomeno proporzionale agli elementi delle tabelle attori del JOIN stesso. Essendo che, ovviamente, non possiamo in alcun modo diminuire le dimensioni dei dataset, quello che progettualmente si incentiva è diminuire il più possibile il numero di JOIN e, per far questo, le parole chiave del modello ROLAP sono proprio "denormalizzazione" e "ridondanza". L'estensione ROLAP in figura "2.7" dell'esempio che abbiamo utilizzato nella discussione MOLAP mette in luce proprio quanto precedentemente discusso:

- Le "Dimensioni" sono rappresentate tramite opportune tabelle, cosiddette "Dimension Table", dove i livelli di aggregazione sono organizzati come attributi della relazione stessa

- Il Fatto da analizzare è rappresentato tramite un'opportuna "Tabella "Vendite", o "Fact Table", la cui chiave è data dall'insieme delle chiavi

di ciascuna Dimension Table, aspetto che rende possibile interfacciarsi direttamente a quest'ultime tramite la specifica chiave

- Le "Misure" sono attributi diretti della Fact Table

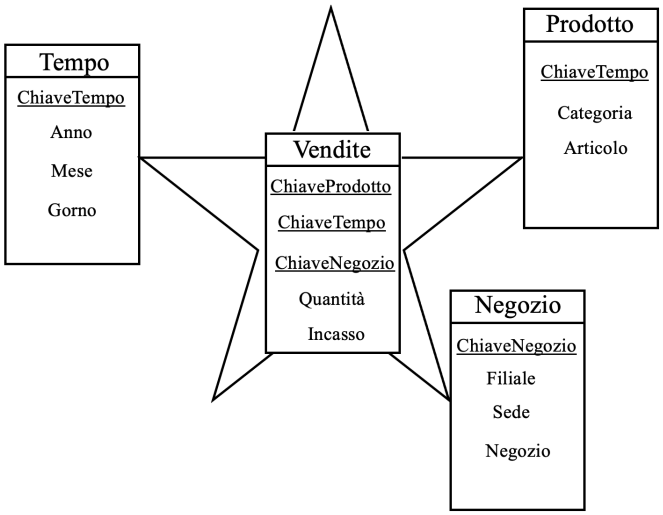

Figura 2.7: "Vendite" in modello ROLAP

La struttura denormalizzata delle Dimension Table risulta abbastanza palese, in quanto i livelli di aggregazione presentano tra loro, per definizione stessa di "granularità", dipendenze funzionali. "Nazione" dipende funzionalmente da "Filiale" e "Filiale" dipende funzionalmente da "Negozio"; questo tipo di dipendenze vengono anche dette "transitive". A differenza delle Dimension Table, la Fact Table è, per costruzione, in forma normale di "Boyce e Codd": rappresentando quest'ultima il Fatto stesso, ovvero le singole vendite di prodotti, la BCNF è fondamentale al garantire il vincolo di unicità delle tuple ed evitare anomalie di inserimento che, all'effettivo, sono le uniche anomalie rilevanti in questo contesto; non dimentichiamoci, infatti, che stiamo studiando un modello di Data Warehouse: modifiche ed eliminazioni sono molto rare, gli inserimenti invece sono ciclici. In un certo senso, BCNF è un controllo ulteriore agli inserimenti che si fanno nella Fact Table, essendo che molto dell'integrità dei dati è garantita a priori dalla fase di "Transform" dei

pattern ETL/ELT che discuteremo successivamente. Come già discusso nella sezione MOLAP, una delle differenze più sostanziali tra questi due modelli è, proprio, "cosa" venga rappresentato: il modello MOLAP non rappresenta "eventi reali" come fa, invece, il modello relazionale, difatti nella Fact Table ROLAP si terranno conto di tutte le vendite singolarmente, mentre nel modello MOLAP si tiene conto delle vendite complessive di un dato prodotto per particolari configurazioni delle dimensioni rimanenti; proprio in questo senso il modello ROLAP non pre - aggrega i dati come MOLAP, ovviando al vincolo di dataset ridotti, ma risponde alle query computazionali effettuando direttamente le operazioni opportune sui dati. Il modello in figura "2.7" viene normalmente definito come "Star Schema", in quanto ricorda proprio la struttura di una "stella" con la Fact Table centrale. Lo "Star Schema" permette di ridurre sostanzialmente il numero di JOIN necessari alle computazioni, si pensi al come tutte le Dimension Table siano "ad un solo passo", un solo JOIN per intenderci, di distanza dalla tabella dei Fatti soggetto delle computazioni. Adottando altri accorgimenti progettuali, come una filosofia colonnare ad esempio, il modello ROLAP costituisce una delle soluzioni, ad oggi, più efficienti per la gestione di grandi volumi di dati e, come suddetto, particolarmente apprezzata in quanto evoluzione dei già adottati modelli relazionali.

In questo capitolo abbiamo discusso dell'importanza rivestita dai database relazionali nel Data Management moderno, introducendo il concetto di Data Warehousing, e della forte necessità, duplice gli ingenti volumi Big Data, di efficientare le computazioni aggregative utili all'analisi dimensionale, adottando logiche denormalizzate e memory management colonnare, discutendo delle limitazioni di altri approcci, altrettanto validi, come quello MOLAP e di come, comunque, si decida di non abbandonare il retaggio relazionale, estendendolo a veri e propri standard che adottino i suddetti meccanismi, come il modello ROLAP e il suo "Star Schema". Nel prossimo capitolo introdurremmo, oltre che il concetto di Cloud Computing, il motore principe del nostro caso di studio: "Google BigQuery", quale ci permetterà di esplorare i "Cloud Data Warehouse" già accennati precedentemente, una delle solu-

zioni più recenti nell'ambito del Big Data Management, e di come, proprio, le tecnologie Cloud permettano di risolvere tutte le cricità di "obsolescenza" dei database già discusse nel "Capitolo 1", mettendo in pratica, d'altronde, tutti i concetti implementativi descritti finora.

3

Cloud Computing

Il “Cloud Computing” è uno degli approcci più moderni alla risoluzione dei problemi di digitalizzazione e implementazione di sistemi informatici nei contesti enterprise. Il Cloud costituisce uno dei motori principali del dinamismo con cui, negli ultimi anni, si è evoluto il settore IT, tanto che è stato eletto come una delle “tecnologie abilitanti” dell'Industria 4.0.
A mio avviso, molto del concetto stesso di “Cloud Computing” è stato trattato nel “Capitolo 1 - La Rivoluzione Big Data”: in settori come quello del “Data Management” si è arrivati alla situazione, quasi paradossale, dove non esistano più validi motivi per fare a meno di tecnologie Cloud based, elevando quest'ultimo alla stregua di uno standard su cui si fondino tutti i sistemi informatici moderni. La vastità delle possibilità che offre il “Cloud Computing” rende particolarmente difficile fornirne una definizione rigorosa, sebbene in letteratura si usi definirlo come: un particolare “modello di erogazione di servizi informatici” principalmente orientato alla costruzione di soluzioni flessibili e “su misura” alle richieste del consumatore. I principali servizi forniti da tecnologie Cloud vanno dallo storage alla potenza di calco-

lo, dai database a infrastrutture di rete... La grande rivoluzione del Cloud permette l'accesso ad un portafoglio, pressoché illimitato, di servizi informatici, sfruttando un modello di consumo "pay as you go", dove al cliente viene addebitato un costo che è direttamente proporzionale all'utilizzo delle risorse, liberandolo dalla necessità di acquistare e gestire in proprio costose infrastrutture hardware e software, definite comunemente "On Premise". Il concetto di fondamentale importanza quando si tratta di "Cloud" è quello di "scalabilità": quando il cliente acquista un servizio, può deciderne i parametri fondamentali e le specifiche, adattandolo alla propria domanda; un semplice esempio è quello dei servizi di "Cloud Storage": provider come iCloud, Google Drive, AWS... permettono tutti di acquistare una repository online dove archiviare i propri file, documenti, foto... e, se necessario, la capacità della repository è estendibile acquistando un piano diverso. La scalabilità delle risorse Cloud permette di costruire soluzioni che siano durature nel tempo, e non molto sensibili al dinamismo del mercato IT. Nel Data Management si sfrutta la scalabilità del Cloud per implementare Data Warehouse che possano adattarsi alle caratteristiche di "Variabilità" e "Volume" dei Big Data, aggiustando dinamicamente le capacità di archiviazione e calcolo per garantire prestazioni efficienti funzionalmente alla particolare richiesta. In figura "3.1" viene presentato il funzionamento generalissimo di "Cloud Computing":

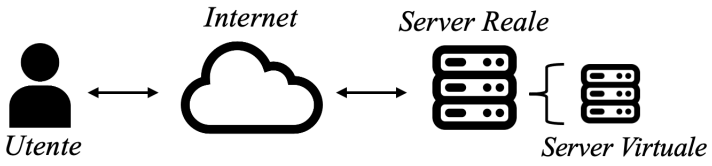

Figura 3.1: Schema Generale Architettura Cloud

L'infrastruttura fisica del provider viene virtualizzata in uno o più server, e viene alloctata sulla base delle risorse di memoria e calcolo concordate con il cliente. Per l'utilizzo e l'amministrazione si accede all'infrastruttura tramite interfacce utente/API. Nella prossima sezione approfondiremo i tre principali modelli di servizio Cloud, discutendo delle caratteristiche e principali

casi d'uso: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS).

3.1 Modelli di Servizio Cloud

I modelli "IaaS", "Paas" e "SaaS" costituiscono differenti modalità di erogazione di servizi Cloud a seconda delle richieste del consumatore. La differenza sostanziale tra i tre, da cui dipendono i differenti casi d'uso, coincide con il particolare "livello di responsabilità" del Cloud provider. "IaaS", acronimo di "Infrastructure as a Service", è il modello più semplice e tradizionale di servizio Cloud, qui il provider si assume solo le responsabilità di gestione dell'infrastruttura, il livello più basso, mettendo a disposizione del cliente core, RAM, storage, CPU... tutte le risorse di memoria e calcolo precedentemente discusse. In "IaaS" il provider deve garantire che l'infrastruttura fornita funzioni correttamente a livello hardware e non ha alcun ruolo in quello che sarà il successivo utilizzo da parte del cliente, che rimane responsabile della configurazione e manutenzione di tutti gli strumenti e le applicazioni che decida di implementarvi. Questo modello di servizio è principalmente adottato quando interessa implementare applicativi "ex novo", sfruttando la scalabilità delle risorse che offre l'Infrastructure as a Service.
All'estremo opposto del modello "IaaS" troviamo il modello "SaaS", o "Software as a Service", dove il Cloud provider ha il maggior livello di responsabilità, fornendo applicazioni completamente sviluppate a cui gli utenti possono accedere tramite Internet. Le applicazioni sono eseguite direttamente sulle infrastrutture dei provider, per cui gli utenti possono solo utilizzare funzionalità implementate dallo stesso e, nonostante sia uno dei servizi Cloud più elargiti, ai fini di ciò che tratteremo non sarà così rilevante.
A cavallo tra "IaaS" e "SaaS" troviamo il modello "PaaS", dove si offrono agli utenti determinate piattaforme software e tool sviluppati direttamente dal provider; quest'ultimo sarà tenuto, quindi, a garantire non solo che l'architettura hardware funzioni correttamente, ma anche che le piattaforme software fornite facciano lo stesso, occupandosi della manutenzione. La dif-

ferenza sostanziale con “SaaS” risiede nel fatto che il “PaaS” fornisce una piattaforma orientata all’implementazione di un determinato servizio, che si struttura sul software e sugli strumenti garantiti dal provider, “SaaS” è puramente orientata all’utilizzo e non allo sviluppo di applicazioni, differisce da “IaaS”, invece, in quanto le piattaforme e gli strumenti di sviluppo non sono direttamente implementati dal consumatore, ma forniti, appunto, dal provider stesso. Il motore principe di questa tesi, Google BigQuery, è un database Cloud-based appartenente alla categoria “PaaS” di modelli di servizio: lavorare con BigQuery vuol dire non prestare attenzione all’architettura che è alla base del database, garantita e gestita da Google, concentrandosi unicamente sul suo utilizzo prettamente applicativo all’interno di processi di ELT ed ETL, o come Data Warehouse in contesti enterprise.

3.2 Infrastructure as Code: Terraform

Come già discusso nelle sezioni precedenti, l’implementazione di Data Warehouse su Cloud è una delle soluzioni ad oggi di più largo utilizzo. Tutte le tecnologie ed i prodotti Cloud based vengono gestiti tramite particolari software comunemente definiti come “Infrastructure as Code” o, più semplicemente, “IaC”. Un IaC è un approccio alla gestione, creazione, configurazione e manutenzione di infrastutture informatiche basato interamente su codice: il vantaggio intrinseco di questo metodo è fornire agli sviluppatori la possibilità di definire concretamente tutte le risorse dell’infrastruttura tramite codice, evitando tutti quei processi manuali che, oltre ad essere tipicamente dispendiosi, sono spesso soggetti ad errori. “Terraform” è uno dei tool IaC più utilizzati nella pratica, permettendo all’utente di gestire sia risorse Cloud che On - Premise tramite opportuni file di configurazione scritti in “HashiCorp Configuration Language - HCL”, sviluppato dall’omonima azienda che detiene la proprietà intellettuale di Terraform. HCL è un linguaggio dichiarativo, particolarmente calzante a quanto abbiamo discusso in quanto, come da fondamenta di tutti i linguaggi dichiarativi, permette di definire lo “stato desiderato”, ovvero cosa si vorrebbe idealmente raggiungere, senza specificare

direttamente tutte le operazioni intermedie al raggiungimento stesso, come farebbe, d'altra parte, un linguaggio imperativo. HCL permette all'utente di limitarsi unicamente a definire tutti i parametri sensibili di una risorsa per configurarla all'effettivo, sfruttando la struttura fondamentale di "blocco", inteso come un insieme di assegnazioni atte alla particolare configurazione di una certa "risorsa", che, tipicamente, comprende più attributi da specificare. Il workflow generalissimo di Terraform può essere facilmente sintetizzato in tre fasi: scrittura, pianificazione e applicazione. La fase di scrittura è quella dove si sviluppa il codice che andrà a definire le risorse. Il codice è organizzato in file HCL, comprensivi della particolare estensione ".tf", a cui si farà accesso in lettura durante la successiva fase di pianificazione. La fase di pianificazione si preoccupa di organizzare in una "lista" tutte le azioni definite in ciascuno dei file .tf predisposti nella fase di scrittura, creando un "piano di esecuzione" comprendete tutte le azioni di creazione, aggiornamento ed eliminazione necessarie; questa fase risponde all'istruzione "terraform plan", impartito tramite linea di comando. L'applicazione coincide, formalmente, nell'esecuzione di ciascuna delle azioni definite nel piano di esecuzione, anche qui tramite il particolare comando "terraform apply".

Figura 3.2: Terraform Provider nel Cloud

Ciò che risulta veramente interessante discutere della fase di applicazione è il "come" Terraform possa accedere alla piattaforma del provider Cloud ed effettuare le suddette azioni: indipendentemente dal provider, sia esso Google, Amazon, Microsoft... (si veda figura 3.2), tutte le informazioni sensibili della propria infrastruttura Cloud, come le chiavi e token di accesso, sono contenuti in un particolare file di configurazione, tipicamente JSON, che è direttamente reperibile dalla piattaforma PaaS stessa. La fase di scrittura impone, quindi, che tutte le risorse vengano definite a valle di un particola-

re “Blocco Provider” che dichiara, appunto, quale sia l’esercente del servizio Cloud e permetta a Terraform di collegarvisi in fase di applicazione; il “Blocco Provider” implementa, sostanzialmente, tutte le operazioni di collegamento con l’API dell’esercente.
Nelle applicazioni di Data Management, Terraform riveste un ruolo di particolare importanza: risulta possibile, infatti, creare, modificare ed eliminare dinamicamente istanze di database SQL, tra cui dataset o tabelle in motori come BigQuery, permettendo di automatizzare il processo di gestione di un Data Warehouse Cloud based.

3.3 Sistemi Distribuiti

Nel “Capitolo 2” abbiamo spiegato in cosa consista il “Data Warehousing” e il suo ruolo in contrapposizione agli RDBMS tradizionali OLTP, nonché sottolineato come contesti enterprise siano, poi, esempi abbastanza emblematici di tutte le criticità sollevate dai Big Data alle architetture tradizionali. La domanda di mercato moderna impone, oltre che la necessità di gestire grandi volumi di dati, velocità e tempi di risposta “Real - Time”, tanto che le intrinseche inefficienze del modello relazionale, nato e progettato per ovviare a necessità molto molto differenti, vengono rese meno vincolanti da accorgimenti come un memory management colonnare, nonché rivisitazioni stesse dello schema relazionale cui ROLAP; quello che, poi, storicamente ha rappresentato la vera e propria singolarità di passaggio dai sistemi tradizionali a quelli moderni è l’adozione di “Architetture Distribuite”, che discuteremo proprio in questo capitolo. I “Sistemi Distribuiti” rappresentano l’ultimissimo tassello necessario a comprendere approfonditamente come si strutturino le moderne architetture Big Data, contestualizzando le infrastrutture dove vengono implementati fisicamente tutti i concetti discussi nel capitolo precedente. Con “Sistemi Distribuiti” si parla, sostanzialmente, di una filosofia di progettazione per sistemi informatici che incentivi prevalentemente scalabilità, flessibilità e performance, tutte proprietà imprescindibili, come accennato precedentemente, per la domanda attuale Big Data. Inizialmente,

i sistemi informatici prediligevano architetture cosiddette “Monolitiche”, ovvero centralizzate in unica macchina che gestisse tutte le richieste, approccio che presentò ben presto gli intrinseci limiti in termini di capacità computazionale, storage e distribuzione del carico di lavoro; un esempio già citato, tra le righe, è la struttura “Mainframe” di un Data Warehouse tradizionale. Il principale vincolo delle architetture “Monolitiche” è legato al concetto di “scalabilità”: generalmente, intendiamo così la possibilità di incrementare le prestazioni del sistema o agendo a livello hardware sul singolo server, spesso definita “scalabilità verticale”, o aumentando il numero di macchine su cui distribuire il lavoro, “scalabilità orizzontale” in letteratura; si pensi a tutte le moderne applicazioni web dove non è possibile predirre a priori quale potrebbe essere il massimo traffico e numero di richieste a cui si sottopone il sistema, qui la scalabilità verticale diventa effettivamente un problema poiché le prestazioni sono limitate a quello che è il tetto hardware disponibile dalla tecnologia moderna, nonché soluzioni tipicamente costose. Il concetto di “Sistema Distribuito” si basa, proprio, sullo sdoganare le architetture “Monolitiche” in favore di “insiemi di macchine”, spesso definiti “clusters”, su cui è possibile distribuire il carico di lavoro, potendo raggiungere un incremento prestazionale teoricamente infinito poiché non limitato da tetti fisici come quello hardware, il che è un vantaggio evidente. Sebbene le architetture distribuite siano, all’effettivo, costituite da diverse macchine comunicanti tra loro, macroscopicamente sembrano comportarsi come un’unica grande entità, dove richieste semplici possono essere processate da nodi diversi del cluster singolarmente, come richieste più complesse possono essere scomposte in unità di lavoro più piccole e processate parallelamente tra più nodi della rete, che è, d’altronde, il principio alla base degli “MPP” (Massive Parallel Processing) come BigQuery.

3.3.1 Cloud Data Warehouses

Il passaggio che c’è tra un’architettura di Data Warehousing distribuita ed una “Cloud Based” in realtà è particolarmente irrisorio, in quanto quest’ultime sono, in buona sostanza, erogazioni di servizi PaaS da parte del provider,

il quale fornisce al cliente una piattaforma con cui possa comunicare con l'infrastruttura distribuita gestita dal provider stesso, ma che si basa, internamente, sugli stessi concetti di un'architettura distribuita proprietaria; l'adozione cloud based ha, però, il grande incentivo di poter usufruire di tutti i vantaggi intrinseci di soluzioni Cloud. Il concetto fondamentale delle architetture distribuite e, di conseguenza, dei "Cloud DW" è la "Disaggregazione tra Computazione e Storage": nelle architetture "Monolitiche" le componenti di memoria ed elaborazione sono parte di uno stesso sistema, distribuire implica, invece, "spalmare" la complessità computazionale richiesta su nodi distinti del cluster, macchine separate quindi, tutti facenti riferimento ad un layer di storage comune gestito indipendentemente dai nodi di calcolo, anch'esso distribuito su più server; sebbene, complessivamente, l'architettura continui a comportarsi come un unico database. A livello prettamente pratico, i nodi di calcolo coincidono con cluster di "Virtual Machine" dei server del provider e i layer di storage sono prevalentemente "Data Lake", come si può apprezzare in figura "3.3".

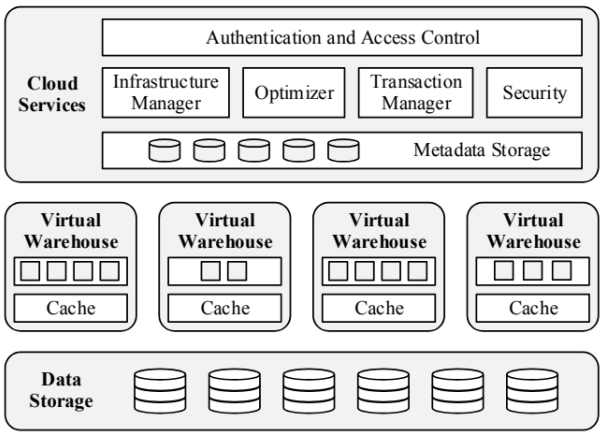

Figura 3.3: Architettura Snowflake Data Warehouse

3.3.1.1 Data Lake

I "Data Lake" sono, in buona sostanza, delle grandi "repository" dove è possibile archiviare qualunque tipologia di dato, sia esso strutturato, semi

strutturato o non strutturato, senza distinzione. I "Data Lake" non permettono di effettuare computazioni che vanno oltre semplici visualizzazioni dei dati, clausole SELECT in termini SQL, per il semplice fatto che non esista un formato predefinito con cui lavorino ed è difficile, proprio per questa eterogeneità dei dati in analisi, implementare DBMS o logiche di alcun tipo. Il concetto alla base dei Data Lake è quello di "metadato": come abbiamo discusso nel precedente capitolo, il principio stesso di database è far si che $dato = informazione$, contestualizzando correttamente opportuni valori in una struttura, come quella relazionale, che fornisca informazioni collocando i dati in altrettanto opportune tabelle, con specifici attributi. Generalmente si definiscono "metadati" tutte quelle quantità che diano informazione sui dati stessi, definendone le caratteristiche e contestualizzandone il contenuto; in un certo senso, gli stessi "attributi" di un modello relazionale sono intendibili come metadati. Proprio grazie a quest'ultimi, i Data Lake riescono a contestualizzare gli unstructured data e a fornire meccanismi di risposta a determinate richieste che li comprendano; il problema fondamentale è come si possano estrarre questi metadati e in questo consiste la nuova frontiera del Data Management: Machine Learning, Natural Language Processing e tutta un'altra serie di applicazioni "AI - Oriented" permettono di automatizzare processi che altrimenti rimarrebbero "manuali", tra questi l'estrazione di metadati. L'infrastruttura e l'innovazione che ha favorito la capillare adozione dei Data Lake ruota attorno al concetto di "File System Distribuito": i dati sono organizzati su diversi dispositivi di archiviazione fisica, collegati tra loro tramite una rete di comunicazione, distribuendo la necessità di storage capacity su cluster di unità d'archiviazione, del tutto analogo a quanto si faccia computazionalmente; i Data Lake sono, quindi, sì delle "repository centralizzate" di storage, tuttavia "distribuite" su cluster di hardware distinti. GFS ("Google File System"), HDFS ("Hadoop Distributed File System"), come GCS ("Google Cloud Storage") ed Amazon S3 sono tutte moderne implementazioni di file system distribuiti. La praticità di questi strumenti sta nella loro scalabilità e nel loro bassissimo costo. Moderni servizi di cloud storage come GCS, ad oggi 2024, contano di tariffe sui 2 centesimi al mese per ogni GB di dato archiviati, permettendo, inoltre, di spostarli in categorie

di storage ad ancor più basso costo se non frequentamente acceduti, cui GCS "Archive Storage" o "AWS Glacier". Tutte queste proprietà rendono i Data Lake soluzioni del tutto appetibili come metodo di archiviazione per qualsiasi architettura moderna di Data Management, in particolare come "Staging Area" per ETL, in dettaglio nel prossimo capitolo. Le ottime prestazioni dei Data Lake, ed i suoi costi particolarmente ridotti, fanno si di poter "centralizzare" tutti i dati all'interno dell'organizzazione, evitando intrinsecamente il concetto di "Data Silos" trattato nell'introduzione al Data Warehousing, qualità indispensabile in contesti enterprise. Il pattern principale degli ultimi anni in ambito Data Warehouse è stato, proprio, quello di disaggregare componente computazionale e di storage dei dati, sfruttando Data Lake come "general purpose storage engine".

3.4 Google BigQuery

BigQuery è il Cloud Data Warehouse nativo di Google. Come suddetto, quest'ultimo coincide nell'erogazione di un servizio "PaaS" da parte del provider, che ne garantisce la capacità computazionale e di storage richiesta. Ciò che ci interessa veramente analizzare in questa sezione è come funzioni "under the hood" un MPP come BigQuery, discutendo con particolare attenzione di "Dremel", il suo query engine. BigQuery come Cloud DW ha fatto, storicamente, di Google un vero e proprio pioniere, spingendo altri prodotti, come Amazon Redshift, a evolversi di conseguenza. Più di una decade di sviluppi ha portato BigQuery a rivoluzionare l'architettura dei Data Warehouse Cloud based, semplificando l'analisi di ingenti quantità di dati anche e soprattutto al personale non tecnico. La "non scalabilità" di SQL si è dimostrata, in realtà, un falso mito degli anni 2000, dove il focus del lavoro di ricerca fu principalmente incentrato nella realizzazione di framework come MapReduce per l'analisi di Big Data; non solo si è dimostrato un falso mito, ma SQL ha continuato a costituire un'alternativa semanticamente valida anche per esprimere computazioni di complessità crescente, esempio primissimo i modelli di Machine Learning: BigQuery ML permette, appunto, la creazio-

ne, validazione ed inferenza direttamente su tabelle BigQuery con semplici primitive dichiarative. BigQuery supporta, d'altronde, transazioni "ACID" e ha forti capacità di Data Governance: i meccanismi di "Row and Column Level Security", che potremmo sintetizzare in un semplice controllo dell'autorizzazione, o meno, dell'utente a visualizzare tali dati, rendono BigQuery uno strumento particolarmente calzante ai contesti enterprise. Tra tutti i servizi Google è, probabilmente, il più integrato: in BigQuery si incentiva totalmente la comunicazione con tutti gli altri servizi forniti non solamente da Google, ma anche da altri provider, nonché si interfacci agevolmente con molti tra gli strumenti di Business Intelligence più utilizzati nella pratica per reporting e analisi dei dati, cui Looker e Looker Data Studio; questo concetto di "integrazione" con altri sistemi sarà, poi, importante oggetto di discussione nelle sezioni successive.

3.4.1 Dremel

Il funzionamento generale di BigQuery si basa, come accennato in precedenza, su "Dremel": query engine di Google in produzione dal 2006 progettato su architettura distribuita, capace di gestire dataset e tabelle sull'ordine dei "PetaByte" (milioni di GigaByte) per ciascun utente. Dremel si è distinto per essere uno dei primissimi sistemi a permettere l'analisi di una così ingente quantità di dati tramite SQL, introducendo 3 innovazioni fondamentali:

1. Memory management "colonnare", secondo gli stessi esatti principi a livello hardware discussi nel "Capitolo 2"

2. Disaggregazione tra "Computazione" e "Storage", coincidente anch'esso con l'adozione di un'architettura distribuita

3. Analisi "in situ" dei dati: in buona sostanza, si evita di far circolare grandi quantità di dati, come vedremo, tra i diversi nodi del cluster e circoscrivendo, in un certo senso, i movimenti di volumi maggiori solo dove l'infrastruttura di rete consente tempi e velocità di trasmissione accettabili

Dremel, come accennato poco fa, sfrutta una struttura ad "albero" organizzata in diversi "Executor Servers" per, appunto, l'esecuzione delle query. L'aspetto più rivoluzionario dell'approccio di Dremel è proprio il dinamismo nella gestione delle query: il processo, che va dall'iniezione della query alla sua esecuzione effettiva, permette di "scalare orizzontalmente" il lavoro interfacciando, tra loro, diverse componenti; quest'ultimo parte dal "Root Node", ovvero il server che gestisce l'interazione con l'interfaccia utente, GCP (Google Cloud Platform) nel nostro caso, a cui vengono inoltrare le query che vi si inseriscono, responsabile, in prima battuta, di indirizzare correttamente la query al successivo livello di "Intermediate Servers" e, in ultima, di rendere i risultati disponibili all'interfaccia utente stessa; la funzionalità fondamentale del "Root Node" è di "traduzione" della query: Dremel è un MPP e gestisce l'esecuzione della query iniziale scindendola in sotto - query tali che ciascuna lavori su partizioni diverse del dataset, o tabelle specifiche, in modo da, proprio, "scalare orizzontalmente" il lavoro. Questo processo viene definito "Query Plan" in letteratura. Gli "Intermediate Servers", o "Mixers", a loro volta traducono le sotto - query ricevute in istruzioni più semplici e "atomiche", mirate al filtraggio ed ottenimento dei dati strettamente necessari alle computazioni, mettendo in evidenza JOIN, GROUP BY, clausole WHERE... tutte risolte dai "Leaf Node", ottenendo, singolarmente per ogni partizione, quei sottoinsiemi del dataset su cui effettuare direttamente le computazioni; più semplicemente: i Leaf Node filtrano i dati sensibili, gli "Intermediate" effettuano le aggregazioni. Leaf ed Intermediate Nodes sfruttano "Jupiter", la rete dei Data Center di Google, quale rende lo scambio anche di grandi volumi di dati estremamente veloce ed efficiente, lavorando a velocità di $1PB/s$ (PetaByte al secondo); da questo possiamo concludere come, in buona sostanza, Leaf e Intermediate Nodes risiedano in server all'interno dei Data Center di Google, non una prerogativa dei Root Node come vedremo. I Leaf Nodes sfruttano Jupiter per l'accesso a "Colossus", il file system colonnare distribuito di Google, dove sono memorizzati fisicamente i dati da computare; quest'ultimi, come poi in qualunque sistema d'archiviazione dati, vengono memorizzati nel comune formato colonnare "ColumnIO". ColumnIO include statistiche generali, come valutazioni preliminari di fun-

zioni di minimo o massimo, prese in considerazione quando strettamente necessario dal Query Plan per risparmiare computazioni sui nodi intermedi, nonché header con puntatori alla location delle varie colonne all'interno del file; concetto di fondamentale importanza in Colossus, che viene proprio garantito da ColumnIO, è la "compressione" dei dati: Colossus utilizza una serie di algoritmi di codifica senza perdita, di relativa complessità, utili a massimizzare l'utilizzo dello spazio di archiviazione, rendendolo, appunto, una delle soluzioni più competitive in contesti Big Data. A livello puramente pratico, i Leaf Nodes consistono in delle "porzioni" della capacità di calcolo di uno o più server, provvedendo all'esecuzione di un numero specifico e pre - impostato di thread, spesso definiti "slots". Uno dei problemi principali, a livello puramente operativo, dei Leaf Nodes fu l'esecuzione delle operazioni di GROUP BY e le, ormai solite, operazioni di JOIN: la difficoltà di esecuzione delle suddette, principalmente relativa ai contenuti quantitativi di RAM disponibili a ciascuno slot, fu tanto invalidante nell'adozione di Dremel che si decise di dedicare dei dischi locali o, comunque, delle componenti fisiche separate da ciascun Leaf Node proprio per permettere l'esecuzione di JOIN e GROUP BY in tempi accettabili, introducendo, però, ulteriore latenza nel processo complessivo di gestione della query; si fa riferimento a tali unità separate o, più generalmente, a questo intero meccanismo tramite l'appelativo "shuffle" o "shuffling". Gli Intermediate Nodes non utilizzano Jupiter per l'accesso a Colossus, bensì per prelevare i dati proprio dai dispositivi di shuffling, effettuando ciascuno le aggregazioni relative alla propria "porzione" del dataset e, successivamente, inoltrando i risultati al Root Node; quest'ultimi sono valori di sintesi singoli, risultato delle aggregazioni, e, tipicamente, in questa "manche" non si parla di grandi volumi di dati da trasmettere, ecco spiegato perché la comunicazione Root/Intermediate Nodes non avviene tassativamente tramite Jupiter. Ottenuti tutti i risultati da ciascun Intermidiate Node, il Root effettuarà un'operazione finale di UNION ALL ottenendo il risultato dell'interrogazione iniziale; si noti come, alla fin fine, il principio alla base di Dremel sia una sorta di "Divide et Impera". Il Dremel "moderno" è frutto di evoluzioni continue per permettere di competere, e poi primiggiare, tra tutte le alternative di "Massively Parallel Processing" su

TeraByte di dati; quando ne si concepì il primissimo prototipo, fu rilasciato su un'infrastruttura dedicata di hardware e dischi di archiviazione specializzati, cercando di spremerne costantemente la massima performance, aspetto che, a lungo andare, avrebbe rivelato la sua insostenibilità con il crescente volume di dati. Il processo di disaggregazione da architetture "Monolitiche" a "Distribuite" e, quindi, orizzontalmente scalabili, fu senza dubbio il motore del successo primissimo di Dremel. L'architettura moderna di BigQuery si articola, come suddetto, in Dremel come proprio query engine, delegandone la "gestione" a "BORG": servizio distribuito di Google per gestire i carichi computazionali sui suoi Data Center, sostanzialmente un gestore di risorse che si occupa di definire, tra le tante, il numero di slot necessari per l'esecuzione Dremel di tale query, prima pre - impostati, permettendo complessivamente una gestione migliore del carico di lavoro. I volumi Big Data implicano, inevitabilmente, la necessità di affidarsi a storage systems che, indipendentemente, debbano sempre lavorare ad alte prestazioni e gestire molteplici operazioni di I/O, soprattutto se si parli di cluster di macchine, necessità che ha portato dapprima all'adozione di Colossus, che conta di ottimi meccanismi di ridondanza, poi alla sua successiva evoluzione, negli ultimi anni d'altronde, in "Capacitor": oltre alle suddette feature di Colossus, garantisce non solo un rapporto di compressione più alto, ma anche di poter filtrare le colonne direttamente sul dato compresso, evitandone una preliminare decompressione notoriamente intensa per la CPU; "Capacitor" è il file format su cui si costruiscono le moderne implementazioni di BigQuery, sebbene il funzionamento rimanga lo stesso discusso con Dremel. In figura "3.4" riportiamo un'esemplificazione grafica dell'architettura di Dremel per una migliore comprensione dei concetti precedentemente discussi:

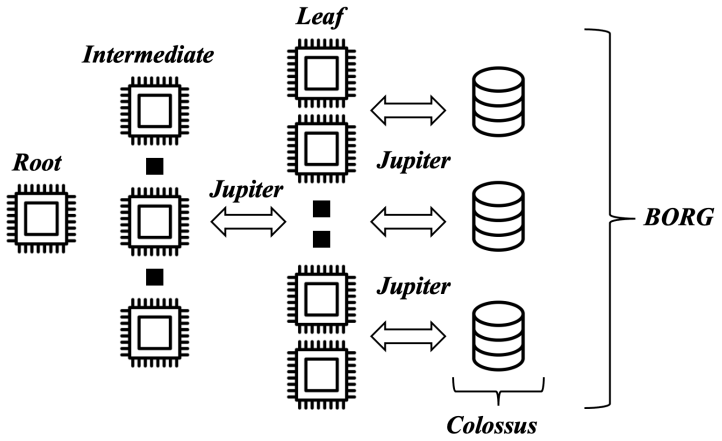

Figura 3.4: Architettura Dremel

3.5 NoSQL e Polyglot Persistence

Sebbene possa sembrare leggermente fuori contesto, discutere del paradigma NoSQL ci permette di introdurre il fondamentale concetto di "Polyglot Persistence", richiamando molti degli aspetti che avevamo espresso prematuramente nel "Capitolo 1". NoSQL è un nome abbastanza evocativo per indicare tutti quei database che decidono di esularsi dall'approccio relazionale in favore di "architetture distribuite" e basate su logiche differenti di gestione del dato. NoSQL è, storicamente, un'alternativa nata proprio per ovviare a tutte le limitazioni del modello relazionale, abbandonando la struttura tabellare in favore di un approccio chiave - valore, prevalentemente "Document Oriented": i dati sono memorizzati in particolari formati semi - strutturati, JSON il più diffuso, come "valori" associati a particolari "chiavi", stringhe per la maggiore, che li identificano univocamente; quest'approccio rende i NoSQL flessbili anche a dati non strutturati, tanto che per quest'ultimi viene definito un particolare tipo di dato "BLOB - Binary Large Object". Il concetto di "Document Oriented" è prevalentemente relativo al come i NoSQL trattino le "entità" relazionali: quest'ultime vengono oggettificate nei suddetti documenti, facendo sì di poter intendere *tupla = documento*, permettendo, inoltre, di poter memorizzare molti più metadati contestualizzanti l'entità stessa all'interno del singolo JSON. Le moderne procedure di Machi-

ne Learning, soprattutto se si pensi ad algoritmi di "Classificazione", fanno sì che dati non strutturati possano essere visualizzati e gestiti tanto facilmente quanto i dati strutturati o semi strutturati automatizzando l'estrazione di metadati, facendo dei NoSQL, come suddetto, un'alternativa del tutto competitiva alla domanda Big Data. Ciò che vogliamo approfondire è, in realtà, quanto già discusso nel "Capitolo 1" parlando, proprio, di unstructured data: la Business Intelligence predilige la generazione di report ed un'analisi real - time di dati strutturati multidimensionalmente, poiché sono questi quelli di stretta rilevanza per il business, sebbene una corretta gestione dei dati non strutturati, duplice la grande mole in contesti aziendali, risulti, comunque, importante anche se esuli dall'aspetto prettamente analitico. Proprio così si spiega una delle tendenze più in voga negli ultimi anni: non esiste una singola tipologia di database che possa soddisfare tutte le esigenze di contesti complessi e fortemente differenziati come quello enterprise, bensì si possono costruire sistemi che interfaccino tra loro logiche di gestone del dato molto differenti, ma che presentino significativi vantaggi a seconda dell'utilizzo prettamente pratico che ne si faccia; il concetto di "Polyglot Persistance" consiste, proprio, nell'integrare diverse tipologie di database, relazionale e NoSQL, all'interno della stessa architettura, permettendo di gestire gli unstructured data con la flessibilità di NoSQL e le analytics con tutti i vantaggi che derivano dagli strumenti ROLAP precedentemente discussi, rappresentando l'equivalente pratico di quel DW moderno e più "totalizzante" nella gestione del dato indipendentemente dalla sua struttura.

4

ETL, ELT ed Airflow

Questo capitolo si propone di approfondire quello che, indubbiamente, è uno dei processi principe del Data Engineering: "Extract, Transformation and Load" o, più semplicemente, ETL. Il rapporto tra ETL e Data Warehouse è un rapporto di forte complementarità, in quanto i processi di ETL sono principalmente orientati a fornire i dati che il DW si occuperà di gestire e presentare all'utente finale; in un certo senso, l'ETL è proprio un presupposto al Data Warehousing stesso e permette, in buona sostanza, a quest'ultimo di lavorare correttamente su un dataset che rispetti la formattazione di riferimento. Il capitolo si propone di studiare, inoltre, l'evoluzione dei processi da ETL ad ELT, analizzandone differenze e necessità fondamentali.

4.1 ETL

I processi di ETL si preoccupano di realizzare a livello pratico tutto ciò di cui si era precedentemente discusso parlando del concetto di "Integrazione" nelle Data Warehouse: se quest'ultime devono risolvere la "Varietà" delle

sorgenti Big Data, servendo da database unificato, si presuppone che possano, in qualche maniera, estrarre i dati necessari dai singoli "silo", verificarne la correttezza di formato ed effettuando, all'occorrenza, opportune operazioni... ovviamente, presupporre che il Data Warehouse faccia tutto questo direttamente implicherebbe non solo "addossarvi" troppa responsabilità, ma soprattutto allontanarsi dal focus prettamente pratico di quest'ultimi: archiviare e computare velocemente grandi quantità di dati; per questo si decide di affidarsi a software esterni che prendano in carico queste necessità da parte dei DW: estraggano i dati dalle sorgenti, ne verifichino la correttezza effettuando trasformazioni se necessarie, e li carichino direttamente nelle Warehouse, rendendoli disponibili. Nella parte progettuale e nel presente capitolo vedremo più in dettaglio tutto questo, sfruttando una tra le piattaforme più utilizzate nella pratica: Apache Airflow. Quando si tratti di processi ETL non vi sono particolari problemi né nel capirne l'intrinseca importanza, né nel comprendere cosa voglia dire "Extract", piuttosto che "Transform" o "Load". Il vero problema risiede nel pensare e strutturare un processo di ETL che sia efficiente al proprio contesto applicativo, in quanto tanto dell'aspetto prestazionale di quest'ultimo dipende dalle condizioni di contorno: sorgenti, tipologie di dato, computazioni, budget, necessità di business... difatti è spesso sconsigliato "riciclare" gli stessi approcci di progettazione di pattern ETL per contesti differenti, in quanto ciascuno performerà in maniera distinta a seconda, proprio, delle suddette condizioni. Spesso, in letteratura, strutturare un processo di ETL comprende anche la progettazione della Data Warehouse essa stessa. In quest'elaborato di tesi esuliamo da questo aspetto, in quanto i più moderni sistemi di Warehousing, come ampiamente trattato, sfruttano architetture cloud PaaS e, in un certo senso, non c'è bisogno di nulla più che sottolinearne gli aspetti che le rendano una scelta efficiente e competitiva per il Big Data Management; per questo ci limiteremo a descrivere il funzionamento dell'ETL "puro", inteso unicamente come concatenazione di processi di "Estrazione", "Trasformazione" e "Caricamento".

4.1.1 Extraction

L'iniziale fase di "Estrazione" comporta, come suggerisce il nome, la raccolta diretta dei dati dai sistemi operazionali; il processo tradizionale di ETL può essere schematizzato, nel suo complesso, come segue in figura "4.1".

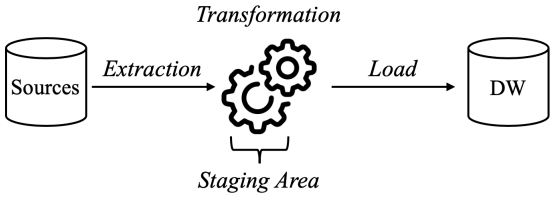

Figura 4.1: Schema Generale ETL

È impossibile comprendere correttamente il processo di "Estrazione" se quest'ultimo venga totalmente esternato da quelle che saranno, poi, le successive fasi di "Transform" e "Load": come vedremo, queste due implicano la necessità di effettuare particolari operazioni sui dati e, per questo, necessitano di un database di appoggio intermedio che definiamo "Staging Area". Il più sostanzioso problema dei pattern ETL è proprio la presenza di quest'ultima, implicante, comunque, un impiego maggiore di risorse; pecca che viene risolta proprio da ELT, come vedremo successivamente. La fase di "Estrazione" comporta, quindi, non solo l'estrazione dei dati direttamente dalle sorgenti operazionali, ma soprattutto garantire il loro corretto caricamento nella "Staging Area", dove saranno successivamente processati. Com'è abbastanza semplice intuire, ad ogni valutazione della fase di "Extraction" non può coincidere un'estrazione della totalità dei dati contenuti all'interno degli operazionali, ciò sarebbe inutile oltre che inefficiente. Bisogna implementare, quindi, una logica che permetta di estrarre solamente i dati "più recenti" rispetto alla valutazione precedente; quest'ultima viene tipicamente realizzata tramite una commistione tra due delle principali soluzioni: "Colonne di Audit" e "Full Diff Compare". La prima, detta anche delle "Colonne di Verifica", impone la realizzazione di uno script back - end che vada a modificare direttamente la logica di memorizzazione degli OLTP, aggiungendo un nuovo attributo temporale coincidente con la data di inserimento nel database di

tale tupla o, comunque, un label che fornisca informazioni affini: tramite quest'ultimo è possibile estrarre facilmente tutti i dati che abbiano attributo temporale pari ad una specifica data come, ad esempio, quella del giorno precedente all'estrazione (SYSDATE - 1)... analoghi ragionamenti si fanno per operazioni di modifica di record preesistenti. La frequenza di valutazione dell'ETL, e della fase di "Extraction" di conseguenza, dipende ovviamente dalle particolari esigenze di business; indipendentemente, le soluzioni software si dimostrano abbastanza semplici da implementare e questa semplicità viene proprio con un sostanziale problema di "robustezza": se l'estrazione venisse, per qualunque motivo, interrotta a mid - process e riniziata successivamente, senza un rapido intervento manuale potrebbero perdersi dati o caricare duplicati nella Staging Area; si pensi, semplicemente, se venisse saltata una valutazione giornaliera del processo di ETL, approcci come SYSDATE - 1 non permetterebbero di estrarre i dati precedentemente tralasciati. "Full Diff Compare", d'altro canto, si basa sul preservare una copia dei dati estratti nella valutazione precedente dell'ETL all'interno della Staging Area, in modo da poter estrarre solamente i dati più recenti effettuando comparazioni record - record; è lampante come questo metodo sia terribilmente dispendioso a livello di memoria e capacità computazionale, ma come sia anche estremamente affidabile: il "Full Diff Compare" permette di rilevare qualunque tipo di variazione nei dati e non è sensibile alle interruzioni o malfunzionamenti come le "colonne di Audit". Nella pratica, tutte le soluzioni moderne si basano su un approccio ibrido tra queste due tecniche, che permetta di compensare, quantomeno parzialmente, le mancanze di ambedue; soluzioni di questo tipo fanno parte della cosiddetta "Estrazione Incrementale". La fase di "Extraction", infatti, si articola in due particolari momenti: "Popolazione del Warehouse" ed "Estrazione Incrementale". La "Popolazione" coincide, fondamentalmente, con la primissima valutazione di un processo di ETL e, tipicamente, è non solo l'estrazione "più semplice", in quanto non è necessario effettuare comparazioni o selezioni particolari essendo il DW destinatario vuoto, ma anche la più dispendiosa, dovendo estrarre la totalità dei dati contenuti negli operazionali; la seconda fase "Incrementale" coincide, invece, con tutti i meccanismi suddetti. Processi di "Extraction" moderni ed

orientati ai volumi "Big Data" preferiscono, spesso, una logica di estrazione "a file": convertire i dati all'interno degli OLTP in formati come "flat file", "XML", "Parquet"... ha molteplici vantaggi, in quanto, molto spesso, con volumi questo tipo il collo di bottiglia ai processi di ETL sono le infrastrutture informatiche su cui i dati stessi dovrebbero viaggiare ed una conversione a file permette, oltre che possibilità di cifratura e, quindi, usufruire di reti pubbliche per il trasferimento, di sfruttare i noti meccanismi di compressione, potendo ridurre dal 30 al 50% il tempo necessario alla fase di "Extraction"; proprio la velocità è, soprattutto ad oggi, la caratteristica principe degli ETL ed i motivi sono, sostanzialmente, gli stessi che ci avevano portato a discutere tutte le alternative più computazionalmente efficienti di database: le analisi devono essere effettuate in tempo reale e su dati costantemente aggiornati.

4.1.2 Transform

La fase di "Transform" comporta l'esecuzione di una serie di operazioni atte al garantire due aspetti sostanziali: struttura e consistenza dei dati caricati nella Staging Area dalla precedente fase di "Extraction". Fondamentale chiarire fin da subito che la fase di "Transform" non può essere vista come la "panacèa" a qualunque tipo di inconsistenza o scarsa qualità del dato, bensì debba garantire che dati di qualità nelle sorgenti operazionali siano di qualità anche nelle Warehouse; il concetto è intrinsecamente questo: non esiste alcun tipo di "Transform" che possa, ad esempio, "sanare" un indirizzo incompleto, una mail inserita in un formato non corretto, campi mancanti se non direttamente ricavabili dalle informazioni in possesso... Per questo è fondamentale che le aziende o, più generalmente, chiunque sia incaricato di raccogliere i dati sul campo, adotti una serie di accorgimenti che permettano di troncare queste problematiche sul nascere: i cosiddetti "Quality Screens", che sono, d'altronde, il cuore concettuale della fase di "Transform" stessa. Per "Quality Screens" si intendono una serie di controlli a cui viene sottoposto il singolo dato per rilevare eventuali inconsistenze ed azioni correttive; molto spesso si tratta di semplici test a logica colonnare: viene verificato se ci siano valori nulli inaspettati, come la violazione di un determinato formato o

valori che cadano al di fuori di un certo intervallo di definizione... i moduli di registrazione ad una determinata piattaforma/sito web sono un esempio del tutto calzante al nostro discorso: gli errori che vengono generati se si provasse a sottomettere, ad esempio, un'età negativa o una mail che non contiene i marcatori fondamentali, come la "@", sono in tutto e per tutto "Quality Screens"; difatti quello che succede nella fase di "Transform" non è poi così differente. Da questo si evince come, in un certo senso, quest'ultima inizi già da una buona "educazione" al trattamento dei dati o, più tecnicamente, "Data Governance". Nella "Staging Area" si riserva una zona di memoria dove vengono caricati i dati che non soddisfano tutti i controlli, per poi processarli successivamente: nella pratica, non si usa caricare il singolo dato non appena un "Quality Screen" fallisca, bensì tutti i dati vengono sottoposti a tutti i controlli progettati in modo da evitare di rilevare inconsistenze parziali nel caso non venga soddisfatto più di un "Quality Screen". Ovviamente, ciascun fallimento implica azioni correttive differenti e, anche qui, è importante adottare opportuni meccanismi che permettano di catalogare ed organizzare correttamente gli errori, in modo tale da rendere più efficiente l'applicazione di azioni correttive: nella pratica, si costruisce un cosiddetto "Error Event Schema", fondamentalmente uno "Star Schema" dove il "Fatto" è l'errore stesso e le "Dimensioni" aggiungono informazioni all'errore, come lo screen che lo ha generato ad esempio, quale permette di identificare con semplicità tutti i record che hanno generato un errore comune, per poi correggerlo; il processo di correzione presuppone che si possa effettivamente correggere le inconsistenze, effettuando computazioni ed operazioni generali che dipendono dal particolare errore generato e dalla logica applicata, mentre in alcuni casi, tipicamente dove la "Data Governance" è abbastanza scarsa, si potrebbe decidere di implementare "Quality Screens" che scartino direttamente i dati con inconsistenze palesi, come quelle citate precedentemente, all'interno della "Transform" ETL stessa. Per i dati che, invece, soddisfano tutti i controlli, si cerca di effettuare operazioni di integrazione e strutturazione nel formato destinatario della Warehouse, queste tipicamente comportano: aggregazioni, unione di dati da sorgenti differenti, aggiunta di attributi e generazione di metadati conformi alla logica di business... queste sono, tipicamente, tutte

operazioni abbastanza semplici che, però, richiedono alla "Staging Area" una capacità computazionale comunque importante, aspetto su cui ritorneremo trattando successivamente della transizione ELT. Si noti come non si sia fatto alcun tipo di riferimento a dati strutturati e non strutturati, rispecchiando tutti i concetti di cui abbiamo parlato il funzionamento generalissimo della fase di "Transform" in un processo di ETL; tuttavia, gli unstructured data necessitano di accorgimenti particolari, che tratteremo in dettaglio nella prossima sezione.

4.1.3 Load

La fase di "Load" comprende, sostanzialmente, la "consegna" dei dati correttamente strutturati alle Warehouse, garantendone il caricamento e la gestione di alcune situazioni particolari. Sebbene possa sembrare la fase più semplice e banale tra tutte, in realtà riveste una fondamentale importanza in quanto le precedenti "Extraction" e "Transform" perdono totalmente di senso se i dati non vengano correttamente caricati e messi a disposizione della Warehouse. Il perché la fase di "Load" sembri apparentemente così semplice è relativo ad un concetto che, nei precedenti capitoli su OLTP, OLAP e Data Warehousing, avevamo già discusso: la "profondità temporale" dei dati; le DW sono pensate per effettuare computazioni efficienti su volumi di dati che, spesso e volentieri, coprono un orizzonte temporale di diversi anni e, proprio per questo, logiche moderne di gestione del dato abbracciano il concetto di "multidimensionalità". In un certo senso, nella disamina su MOLAP e ROLAP abbiamo peccato, e non poco, di presunzione: non si è mai esplorata l'ipotesi, assolutamente realistica, che ci possa essere una certa dimensione, di uno stesso dato, che possa cambiare nel tempo. Riprendendo lo stesso esempio dei capitoli precedenti, supponiamo che la dimensione "Prodotto" presenti un attributo "Prezzo", direttamente ricavabile dagli attributi "Incasso" e "Quantità" del Fatto "Vendita" d'altronde, e supponiamo di analizzare, tramite la chiave "Codice Prodotto", due eventi di vendita distinti: se quest'ultimi presentassero attributi "Prezzo" differenti, la situazione non sarebbe così sconvolgente, in quanto i prezzi di un singolo prodotto tendono a

variare cospicue volte durante l'anno; questo fenomeno viene definito in gergo "Slowly Changing Dimensions - SCD". Sebbene la gestione di questo tipo di situazioni sia di una semplicità disarmante nei database operazionali, in quanto relativa alla gestione di istruzioni analoghe ad "UPDATE" SQL per la logica relazionale, il discorso non è altrettanto banale nelle Warehouse ed il perché risiede nel concetto di "Estrazione Incrementale": valutazioni successive della fase di "Extraction" trattano un record soggetto precedentemente a modifica come se fosse stato inserito recentemente e, quindi, come uno dei "most recent data" effettivamente estratti; se quest'ultimo aspetto vale, e vale, ciò vorrebbe dire che, adottando una logica ROLAP ad esempio, la fase di "Load" fallirebbe a priori in quanto si starebbe cercando, indipendentemente dalla modifica, di inserire nuovamente lo stesso record e, quindi, permettere valori duplicati delle chiavi. Per ovviare a questo problema, i modelli ROLAP adottano particolari tipologie di chiave dette "Surrogate Key": sostanzialmente interi che vengono assegnati al singolo record progressivamente e che lo definiscono univocamente, eleggibili appunto come chiavi; le "Surrogate Key" sono aggiunte dirette della fase di "Transform", effettuate all'interno della "Staging Area" stessa, e fanno parte di tutte quelle operazioni di strutturazione e integrazione dei dati già discusse precedentemente. Chiavi come il "Codice Prodotto" sono attributi che identificano univocamente l'entità in questione anche nella "realtà" al di fuori del database e, per questo, vengono spesso definite "Natural Keys": il fenomeno delle SCD impone che la funzionalità di chiave venga scorporata dalla "Natural Key" in un intero che viene assegnato automaticamente, sfruttando logiche molto simili a contatori, tramite una serie di operazioni comunemente definite come "Surrogate Key Pipeline"; il meccanismo delle chiavi surrogate permette di tenere traccia dell'evoluzione nel tempo di tutte le dimensioni di un determinato fatto e questo chiarisce, ancor di più, il concetto di "profondità temporale" dei dati in un DW. Tutte le considerazioni precedentemente effettuate si sono basate sulla supposizione che una certa dimensione, come il "Prezzo" di un prodotto, vari in modo significativo per il business ed, in realtà, non è sempre così: casi particolari di SCD sono tutti quegli "UPDATE" operazionali mirati alla correzione di record precedentemente inseriti, validi per i "Quality

Screen" ma presenti, ad esempio, di errori di battitura; in questo caso, com'è abbastanza facile intendere, tenere traccia del record errato precedente non è, in alcuna maniera, rilevante e, ovviamente, le "Surrogate Key" permetterebbero assolutamente di aggiungere il record corretto come nuova tupla, ma, a quel punto, come si potrebbe distinguere il record corretto da quello errato? Questo stesso problema, in una forma leggermente differente, si presenta anche per SCD significative come la variazione di "Prezzo": come è possibile distinguere il record più recente da quello meno recente? Richieste di questo tipo sono, in molti casi, assolutamente critiche; da qui in avanti faremo riferimento con "SCD tipo 2" al primo caso che abbiamo analizzato e con "SCD tipo 1" a tutte quelle variazioni non significative. Una soluzione possibile per le "SCD tipo 1" è estendere lo stesso "UPDATE" degli operazionali anche ai Data Warehouse, difatti si fa spesso riferimento a "tipo 1 - Overwrite": si localizza l'attributo modificato e si sovrascrive il relativo record errato all'interno del DW, logica totalmente analoga alle istruzioni di modifica. Gli "Overwrite" vengono sempre e comunque disincentivati in quanto "performance killer": la richiesta di modifica non viene delegata alla "Staging Area", bensì viene presa in carico ed effettuata all'interno del Warehouse stesso, terribilmente inefficiente per questo tipo di operazioni in quanto a logica colonnare. Per ovviare a questo problema, si decide di risolvere anche tutte le criticità della tipo 2 "ibridandola" al tipo 1: indipendentemente che la modifica operazionale sia o meno rilevante per il business, si decide comunque di sfruttare i meccanismi offerti dalle chiavi surrogate ed inserire il record modificato come nuova tupla, provvedendo all'aggiunta di due nuovi attributi "timestamp" analoghi alle già discusse "colonne di Audit" per la fase di estrazione; il primo attributo "timestamp start" indica la data di inserimento del record corrente nel DW, mentre il secondo "timestamp end" indica la data di inserimento di una versione aggiornata del record stesso, relativo a modifiche negli operazionali ed, ovviamente, nullo nel caso queste non siano state effettuate. In questo modo si riesce ad evitare di passare per un "Overwrite" diretto e, sia per SCD tipo 1 o 2, è possibile distinguere chiaramente un record più recente da un record meno recente grazie al che "timestamp end" sia nullo o meno; gli attributi "timestamp" sono aggiunti

direttamente all'interno della "Staging Area". Questo tipo di soluzione comporta, inevitabilmente, un impiego di memoria eccessivo, in quanto vengono caricate nuove tuple anche per tutti quei casi dove non ve ne si traggono direttamente informazioni significative: particolarmente utilizzata, infatti, per implementazioni Cloud based. La fase di "Load" inserisce direttamente, e senza alcun tipo di criticità, nuovi record nel Warehouse che non siano relativi ad operazioni di modifica, altrimenti applicherà tutti gli accorgimenti precedentemente discussi per gestire questo tipo di situazioni particolari. Si noti come tutta la nostra discussione ha avuto una forte matrice relazionale in quanto ci concentriamo principalmente sui sistemi ROLAP, tuttavia è possibile estendere quanto detto anche ai modelli MOLAP ed approfondire ancor meglio tutte le criticità già discusse nella sezione relativa: progettare un pattern ETL per modelli MOLAP ha una complessità sulla fase di "Transform" e "Load" ancora maggiore della controparte ROLAP, in quanto situazioni come "SCD tipo 1 - Overwrite" comportano il ricalcolo, parziale o meno, di tutte le aggregazioni per ogni granularità delle dimensioni e proprio quest'eccessiva dispendiosità a livello di risorse comporta inevitabilmente una complicazione del modello, rendendolo più vincolante e meno reattivo.

4.2 ELT

I Data Warehouse massivamente scalabili su architettura distribuita, introdotti con particolare attenzione a BigQuery nel precedente capitolo, sono le tecnologie abilitanti di un nuovo pattern di integrazione del dato: ELT - Extraction, Load e Transform, di cui parleremo in questa sezione. Sebbene cambiare la disposizione delle lettere da ETL a ELT non costi nulla, in realtà è sintomatico di un profondo cambiamento che c'è tra le architetture tradizionali ed implementazioni moderne, come BigQuery, di Data Warehouse. Il concetto alla base di ELT è principalmente relativo ad un'ottimizzazione del pattern ETL che viene, proprio, dalle ingenti risorse computazionali e di storage dei Cloud DW: tutto ciò di cui abbiamo discusso in questa tesi, cui il modello ROLAP, le architetture distribuite, il concetto di MPP, è vo-

lutamente incentrato all'efficientare l'esecuzione di interrogazioni analitiche, sebbene perda di senso se, all'interno della pipeline del dato stesso, siano presenti evidenti colli di bottiglia, cui la "Staging Area"; i problemi di un database tradizionale nella gestione Big Data sono gli stessi che incontrano le "Staging Area" nell'esecuzione del processo di ETL: limitatezza di storage e risorse computazionali, con evidenti problemi nella gestione di picchi di lavoro in quanto non intrinsecamente scalabili. L'adozione di strumenti come BigQuery, o meglio, la massiva transizione al Cloud Data Warehousing ha fatto sì di far collassare le "Staging Area" direttamente all'interno del layer storage dei Cloud DW e sfruttare la grande capacità computazionale di questi strumenti anche per efficientare i processi di Transform, con la possibilità di implementare alternative anche più costose. I moduli di cui abbiamo discusso, cui generazione delle chiavi surrogate, gestione della duplicazione, monitoring delle trasformazioni... rimangono inalterate, ma sono, adesso, gestite completamente all'intero del Data Warehouse ed orchestrate da software specializzato e sviluppato in SQL. Sebbene quest'ultimo non sia un linguaggio "general purpose", né il più adatto per costruire software, gli use case specifici del Data Warehousing fanno si che, nella pratica, definire tramite SQL tutte le trasformazoni viste in ETL sia un'alternativa del tutto percorribile, sebbene non offra la facilità, prevalentemente in termini di modularità e testing, che si abbia nel trattare problemi di trasformazione complessi con tool più moderni come Pandas o Spark. Spesso i software ETL vengono distribuiti sul mercato tramite pacchetti integrati ed offrenti una serie di funzionalità e trasformazioni predefinite, d'altro canto l'alternativa ELT permette di definire le trasformazioni, come suddetto, direttamente all'interno del Data Warehouse, rendendo molto più semplice aggiungere od eliminare nuove trasformazioni, nonché di modificarne esistenti senza intaccare in alcun modo il flusso di lavoro. Quest'ultimo aspetto è radicalmente più complicato in ETL, in quanto, per l'appunto, pacchetti software integrati e difficilmente adattabili a nuove esigenze, dove è assolutamente possibile implementare nuove trasformazioni, ma è tipicamente molto più complesso ed oneroso farlo. Proprio quest'ultimo concetto, che fa riferimento all'estensibilità dei pattern ELT, è di fondamentale importanza, in quanto, contrariamente a quello che,

poi, storicamente è stato, bisogna cercare di costruire sistemi che possano adattarsi il più possibile semplicemente alle esigenze future e la facilità con cui i pattern ELT possono integrare nuove sorgenti, implementando nuove trasformazioni, è quello che ne fa, poi, la grande fortuna.

4.3 Airflow

Come abbiamo discusso nella precedente sezione, i pattern ETL/ELT sono costituiti dai tre processi fondamentali di "Estrazione", "Trasformazione" e "Caricamento", il quale ordine è relativo al se si adotti, propio, un ETL od un ELT. Ciascuno di questi processi è costituito da sotto - unità di lavoro, o "Task" in letteratura, che definiscono le operazioni effettive che, macroscopicamente, formano il processo stesso, ovvero i singoli spostamenti o caricamenti di file, come le specifiche trasformazioni... "Apache Airflow" è uno dei tool più utilizzati nella pratica proprio per la coordinazione e l'esecuzione delle suddette operazioni, tanto che venga definito come "orchestratore": così si fa riferimento al coordinamento di diverse attività funzionale all'eseguire un flusso di lavoro o un processo nella sua interezza e che, in questo caso, per noi è il pattern ETL/ELT stesso. Airflow è un software open source distintosi per la grande facilità nel gestire pipeline di dati, anche quando i pattern ETL/ELT che orchestra sono particolarmente complessi in termini di task, fornendo una web UI che permetta di visualizzare non solo lo stato corrente di processamento del pattern, ma soprattutto le dipendenze tra i singoli task; alla fin fine, una pipeline di dati e, di conseguenza, l'esecuzione di un pattern ETL/ELT potrebbe essere tranquillamente visualizzato tramite l'utilizzo di DAG, o "Direct Acyclic Graph", tali per cui: ogni nodo sia un singolo task, o gruppo di task, e gli archi siano le dipendenze tra gli stessi a livello di esecuzione. Il concetto di DAG rende particolarmente efficiente non solo la visualizzazione di quale task sia in processamento a tale specifico momento, ma soprattutto rende non ambigua l'esecuzione della pipeline essa stessa, fornendo, d'altronde, la possibilità di implementare task che richiamino ulteriori DAG, tanto che l'esecuzione del pattern è modulabile a diversi livelli di gra-

nularità; potremmo sempre costruire, infatti, un DAG formato unicamente dai tre gruppi di task "Extraction", "Transform" e "Load", opportunamente posizionate. Airflow permette di definire tramite codice Python sia DAG che task sfruttando opportune librerie:

- Il package "airflow.models.dag" implementa un context manager, best practice di Python, che permetta di gestire automaticamente apertura e chiusura del DAG se istanziato con "with" statement. L'esecuzione di un DAG può essere pianificata tramite un parametro "scheduler", che prende un valore "datetime" o una funzione che ritorna lo stesso, in quanto, spesso, nella pratica pattern ETL/ELT vengo eseguiti giornalmente o, comunque, ad intervalli regolari, sebbene la possibilità di unscheduling o trigger manuale venga comunque fornita

- Il package "airflow.operators" permette di usufruire di un parco molto vasto di operatori utili ad effettuare le operazioni più comuni, come il trasferimento di file da GCS a BigQuery (GCSToBigQueryOperator), messo a disposizione da Google stesso, o l'esecuzione di una particolare query da parte di quest'ultimo (BigQueryInsertJobOperator); nella fase "Transform", e particolarmente in pattern ELT, le singole operazioni di trasformazione sono spesso definite tramite query SQL, in questo senso "BigQueryInsertJobOperator" permette di creare task direttamente da file ".sql". È anche possibile definire operatori propri e personalizzabili, semplicemente estendendo la classe "BaseOperator" secondo gli stessi meccanismi standard di "ereditarietà" nei linguaggi OOP. Gli "operators" sono fondamentali per la creazione di task, le cui dipendenze sono dichiarabili tramite l'operatore Pyhton di "shift", nonché risulti possibili definire dipendenze multiple aggregando i task in una lista

Airflow implementa, d'altronde, best practice per la gestione dei log creati dai vari operator, permettendo di, individuato e risolto l'errore, riprendere l'esecuzione corrente dal punto in cui il task è fallito, triggerando il DAG manualmente dalla GUI, facilitandone il troubleshooting.

4.3.1 Architettura

A livello puramente architetturale, le componenti principali di Airflow si articolano in:

- Web App: la UI già accennata precedentemente, utile per la visualizzazione del DAG stesso e per la sua esecuzione, fornendo, come suddetto, possibilità anche di trigger manuale. L'architettura moderna della web app di Airflow è costituita da un front - end scritto in ReactJS e un back - end nel framework Flask di Python, sposando le nuove tendenze di "Single Page Application": l'aggiornamento dei contenuti non è relativo a refresh completi dell'HTML, bensì ad un back - end più solido che li aggiorni dinamicamente

- Database: Airflow ha, ovviamente, bisogno di un DB dove salvare tutti i dati e metadati relativi alla normale esecuzione di Airflow stesso, oltre che necessario alla UI. Quest'ultimo è un relazionale OLTP, cui in produzione si raccomandino spesso MySQL o Postgres, sebbene sia possibile configurare DB più piccoli in locale per sviluppo e debugging

- Scheduler: componente principale di Airflow, si occupa di processare i file Python redatti dagli sviluppatori e costruire tutti gli oggetti e le classi che verranno, poi, mappate nel database relazionale discusso al punto precedente; per realizzare ciò, Airflow utilizza "SQLAlchemy", un "ORM - Object Relational Mapper" per mappare le classi Python in strutture relazionali, concentrandosi prevalentemente sui "metadati": sebbene i file di definizione di DAG e Task siano scritti in Python, le informazioni essenziali su questi, come nomi, versioni e dipendenze vengono memorizzate nel database, analogamente succede per informazioni a runtime come stato, durata dell'esecuzione, risultati e log di errore. Questo approccio permette ad Airflow di monitorare e gestire l'esecuzione dei task in modo efficiente, nonché sia, poi, comprensivo anche di tutte le informazioni sulla schedulazione e trigger dei DAG; quest'ultimi parametri sono configurabili nel file "ariflow.cfg", permettendo allo

scheduler di attivarsi ad intervalli regolari. L'aspetto di fondamentale importanza è la sinergia tra quest'ultimo e i cosiddetti "worker" o "Executors" di Airflow: lo Scheduler si preoccuperà di creare istanze di DAG e task inserendole in una coda software gestita dall'esecutore, che si preoccuperà del processamento effettivo. Dopo essersi attivato, lo Scheduler percorre ricorsivamente tutti i file della folder configurata come root al file system cui punta e processa tutti quelli che terminino con suffiso ".py": se all'interno trova una definizione di DAG, allora procede alla costruzione del DAG stesso e dei suoi task, no altrimenti; le due operazioni di "ricerca" dei ".py" e di effettiva implementazione vengono definite come "crawling" e "parsing" in letteratura

- Executor: quest'ultimo potrebbe essere idealmente visto sia come una "proprietà" od estensione dello Scheduler, sia come una componente software distinta. Come suddetto, il concetto di "Executor" è quello di prendere in carico le task accodate dallo Scheduler ed eseguirle; una delle modalità d'esecuzione più diffuse implementa gli Executor su architetture distribuite, tra cui l'Executor di Kubernetes che discuteremo successivamente

- Triggerer: componente software di Airflow che gestisce la possibilità di eseguire task in maniera asincrona, a seconda di particolari condizioni di, appunto, "trigger". La necessità di un'esecuzione asincrona dipende principalmente dalla complessità degli applicativi reali, in cui si realizzano centinaia di DAG ciascuno comprensivo di altrettante task e in cui, quindi, si incontrano presto dei limiti prettamente infrastrutturali al numero di task eseguibili concorrentemente; i trigger vengono realizzati tramite particolari operatori, detti "Sensor", che aspettano in stato di running l'esecuzione di una condizione prima di abilitare l'Executor

4.3.2 Composer

Airflow, come abbiamo avuto modo di apprezzare nel precedente paragrafo, presenta un'architettura software non monolitica composta di diversi moduli

Python, ciascuno protagonista di un differente passo del workflow complessivo. I maggiori Cloud providers, come descritto nel "Capitolo 3", erogano spesso servizi SaaS che nascondono le complicazioni dell'infrastruttura hardware e di deploy sottostante, impacchettando il software in servizi facilmente gestibili e configurabili dagli user, pronti all'uso dopo pochissimi step di configurazione; questo è il caso di Composer, il servizio gestito di Google per Apache Airflow, e anche di Amazon Managed Workflows For Apache Airflow (AMWAA) per il provider AWS. Composer rilascia le varie componenti Airflow su infrastruttura Google Cloud dedicata, in particolare usa tecnologie di containerizzazione e orchestrazione di container usando "Kubernetes" e GKE, il servizio che implementa un cluster Kubernetes, Cloud SQL come data store e database, GCS per configurazione di DAG, logging e plugin; per "container" si intendono unità software contenenti tutto il necessario per eseguire una data applicazione. Discutiamo più in dettaglio nei seguenti paragrafi di come i vari componenti architetturali di Airflow vengano mappati all'interno dell'architettura Google.

4.3.2.1 Cluster GKE

La componente principale dell'architettura di Google Cloud Composer è il cluster GKE (Google Kubernetes Engine) in cui sono implementate le componenti applicative di Airflow descritte nella sezione precedente, ovvero dove lavorano scheduler, web UI... GKE è un servizio di Google che permette di rilasciare un cluster di VM con al loro interno le componenti di Kubernetes preinstallate, cui api-server, controller-manager, cloud-controller-manager, kube-let per i nodi... Discutere approfonditamente anche di Kubernetes esula dagli scopi di questa tesi, sebbene potremmo intenderlo come un particolare software che permetta di gestire efficacemente la configurazione, il deployment e il funzionamento di servizi containerizzati, garantendo, tra tutte, disponibilità ed allineamento: quando, per cause di forza maggiore cui crash, problemi hardware o malfunzionamenti, uno o più container si discostino dal normale stato di funzionamento, Kubernetes interviene autonomamente per "riallinearli", potendo riavviare i container o ridistribuirli

su altri nodi, garantendo, come suddetto, la disponibilità di ogni componente. Relativamente a quest'ultimo aspetto, Kubernetes usa il concetto di "Pod" come unità di servizio rilasciato sul cluster, ciascuno può contenere diversi container e tutti condividono le risorse del nodo su cui vengono eseguiti. Kubernetes garantisce una certa "ridondanza" nell'esecuzione dei "Pod" su diversi nodi del cluster, garantendo che, se qualcuno di essi fallisca, complessivamente l'applicativo continui ad operare normalmente basandosi sull'esecuzione in "background" di uno dei Pod replica, riavviando o ridistribuendo, come suddetto, i Pod falliti e mantenendo un numero di repliche attive costanti; quest'ultimo concetto è il "deployment" a cui facevamo riferimento precedentemente e, nel caso di Airflow, "Scheduler" e "Web Server" sono rilasciate così. Lo Scheduler, nell'attuazione dei suoi compiti, effettua "crawling" e "parsing" dei DAG definiti nei file Python, costruisce i task e li appoggia in una coda Redis, il tutto all'interno del cluster. Nel suo utilizzo più come, Redis è un database NoSQL utilizzato come cache chiave/valore per un accesso distribuito a bassa latenza da parte di diversi applicativi, definito spesso "in memory" poiché, proprio per garantire migliori prestazioni di I/O, le coppie chiave/valore vengono archiviate in memoria centrale (RAM o DRAM); in questo specifico caso, è utilizzato come una coda pub/sub a bassa latenza da cui i worker Airflow ("Executors") prendono in carico le attività da eseguire, pubblicate dallo Scheduler. Gli "Executors" sono implementati tramite cosidette "Custom Resource" all'interno del cluster Kubernetes: esse sono essenzialmente estensioni dell'API di Kubernetes che, tramite scrittura di software custom, permettono di assumere un controllo maggiore in caso di particolari esigenze operative sul cluster. Composer può essere rilasciato in due configurazioni network: privato o pubblico Nel cluster pubblico l'accesso al database nel "progetto tenant" è inoltrato dal cloudsqlproxy installato in un container sul cluster composer, questo permette di accedere al database semplicemente connettendosi al proxy su quella macchina come se fosse il DB. Nella modalità pubblica, il web server è esposto su internet tramite Ingress GKE, questo di fatto crea un Google Cloud Load Balancer nel progetto che fa da proxy per il traffico web verso il microservizio, separando la componente di accesso e mapping network dall'applicativo, che magari nel suo

ciclo di vita può essere interrotto e ripristinato su macchine virtuali diverse. Nel cluster privato, invece, la creazione del Google Cloud Load Balancer non avviene e il web server è di fatto irraggiungibile dalla rete internet pubblica. Si possono, poi, definire delle "Master Authorized Newtork" per accedere al cluster Kubernetes e al web server, oppure una best practice è quella di creare una macchina virtuale (cosi detta "Bastion Host") sulla stessa subnet del cluster GKE che ne fa da proxy secondo gli stessi meccanismi.

4.3.2.2 Google Cloud Storage

Quando si crea un ambiente Composer, Google crea un "Bucket GCS" all'interno del progetto, contenente le seguenti folders: dags, plugins, data e logs. I microservizi che implementano Airflow all'interno del cluster GKE montano la folder come se fosse un file system attraverso il pacchetto open source "Google Cloud Storage FUSE": quest'ultimo dà l'illusione ai vari Pod che lo storage sottostante sia, all'effettivo, un file system e permettendo alle varie componenti di accedervi senza cambiare il codice sorgente di Airflow; tale necessità viene dal che implementazioni moderne di GCS, basatesi su Data Lake, lo rendono molto più simile ad una mappatura chiave/valore che ad un file system nel senso tradizionale del termine. In particolare, come possiamo vedere in figura "4.2", composer usa GCS per DAG, plugins, e logs.

Folders in the Cloud Storage bucket

Folder	Storage path	Mapped directory	Description
DAG	`gs://bucket-name/dags`	`/home/airflow/gcs/dags`	Stores the DAGs for your environment. Only the DAGs in this folder are scheduled for your environment.
Plugins	`gs://bucket-name/plugins`	`/home/airflow/gcs/plugins`	Stores your custom plugins, such as custom in-house Airflow operators, hooks, sensors, or interfaces.
Data	`gs://bucket-name/data`	`/home/airflow/gcs/data`	Stores the data that tasks produce and use. This folder is mounted on all worker nodes.
Logs	`gs://bucket-name/logs`	Stores the Airflow logs for tasks. Logs are also available in the Airflow web interface and in **Logs Tab** in **Cloud Composer UI**.	

Figura 4.2: Composer - GCS

Questa scelta comporta una serie di vantaggi in termini anche di praticità di rilascio: quando si vuole aggiungere o modificare task o DAG in Airflow, visto che lo "stato di verità" è salvato in GCS, possiamo implementare delle semplici Continuous Integration/Continuous Deployment che automatizzano l'aggiornamento dei file all'interno della cartella "dags" in GCS; sarà poi la componente di Scheduler, a intervalli configurabili, che prenderà e costruirà il DAG da visualizzare poi in UI.

4.3.2.3 CloudSQL

CloudSQL è il servizio Google gestito che facilita deploy, configurazione e mantenimento di un database OLTP relazionale su infrastruttura Google Cloud. CloudSQL supporta al momento tre database: MySQL, PostgresSQL e SQLServer. Composer, in particolare, usa CloudSQL per PostgresSQL come database e, come accennato precedentemente, usa un "progetto tenant" dove rilasciarlo: questo significa che all'interno del progetto Google dove è configurato l'ambiente, l'istanza CloudSQL non è direttamente visibile, sebbene il progetto sia completamente gestito da Google stesso ed è sua responsabilità provvedere ad un sizing adeguato e alla connettività funzionale con il progetto di appartenenza su cui è rilasciato l'ambiente Composer. Per contesti critici, come quello enterprise, è opportuno configurare CloudSQL in modalità HA (High Availability): la configurazione HA propaga le operazioni di scrittura ad una seconda istanza di CloudSQL configurata in una zona diversa e che, nel caso di eventi rari come interruzioni di una zona Google Cloud, provveda a implementare "Fault Tolerance" subentrando all'istanza principale; una possibile configurazione può comprendere anche una gestione automatica dei backup scaricati direttamente in Google Cloud Storage.

5

POC: Pipeline degli esiti di campagne di Marketing

Nei capitoli precedenti si è descritto il contesto tecnologico del settore Big Data, con particolare enfasi sulle problematiche delle architetture tradizionali e come queste vengano risolte convergendo verso le moderne tecnologie di Cloud Data Warehousing; abbiamo, inoltre, discusso di particolari implementazioni di quest'ultime, cui BigQuery, nonché dell'importanza di pattern ETL/ELT come trasformazioni massive del dato in questo contesto. La mia esperienza di tirocinio presso "Accenture S.p.A." mi ha dato la possibilità di lavorare, seppur brevemente, in uno dei progetti per un importante cliente nel settore finanziario, da qui in poi "la Banca", tra le Fortune 500 Global. I sistemi informativi bancari sono estremamente complessi e comprensivi di diversi strumenti che necessitano di interfacciarsi tra loro efficacemente, spesso il problema principe in contesti come questo è, proprio, integrare molti dei suddetti sistemi, obsoleti e sviluppati da fornitori terzi decine di anni fa, con quelli più moderni. La Banca utilizza Salesforce, ad oggi il SaaS CRMS (Customer Relationship Management Service) più utilizzato al mondo: un

Customer Relationship Management Service è uno strumento fondamentale per le aziende moderne, soprattutto di grandi dimensioni, facente parte dei tool che guidano le imprese in tutto il processo di gestione del cliente, dall'acquisizione fino all'analisi del "churn rate" (quanti clienti abbandonino i servizi), offrendo funzionalità di analisi integrate tramite comode web UIs. La Banca si appoggia a "Salesforce Marketing Cloud" per la gestione e creazione di campagne di marketing, interfacciandosi successivamente con "Salesforce Automation", un altro dei tool di casa Salesforce, per implementare processi di trasformazione in SQL sui dati attinti da "Marketing Cloud" ed esportarli in file CSV; la necessità di quest'ultimi è relativa ad uno dei software rendicontativi della Banca, che valuta le campagne di marketing effettuate lavorando, proprio, sui suddetti CSV. Quest'ultimi vengono resi disponibili dalla mattina al rendicontativo, passando, dapprima, attraverso la valutazione di un pattern ELT tramite batch notturni, coincidente con ciò che debba realizzare, all'effettivo, la pipeline in questione. Il collo di bottiglia in questo processo è costituito, proprio, da "Salesforce Automation", le quali prestazioni facevano sì di non solo fallire frequentemente il processo, ma ritardare di molto la disponibilità dei dati: il batch notturno spesso eccedeva la durata prevista dal rendicontativo, introducendo ritardi oltre che parecchi ticket dagli operatori umani. Il progetto consiste nel ristrutturare questo processo in una nuova pipeline basatasi sulle tecnologie Google Cloud, sostituendo "Salesforce Automation" con un ELT orchestrato tramite Composer: i file vengono estratti da "Marketing Cloud" e spostati su Cloud Storage, iniettati in un dataset BigQuery, trasformati nel formato richiesto dal sistema rendicontativo ed esportati in CSV nuovamente su GCS, rendendoli disponibili a quest'ultimo. Per motivi di proprietà intellettuale, mostrerò una procedura semplificata su dati anonimizzati, sebbene il software sviluppato per la presente POC sia pienamente conforme alla realizzazione del progetto originario; nel corso della discussione, mi soffermerò solamente sulle porzioni di codice più importanti, il versionamento completo è consultabile sulla repository GitHub di cui il link in Appendice.

5.1 Gestione delle Risorse Google Cloud con Terraform

La gestione delle risorse progettuali è delegata a Terraform, già introdotto nel "Capitolo 3". Prima dell'infrastruttura, è necessario definire dei parametri di configurazione indicanti lo "stato di verità" della stessa: in ambiente Google Cloud, la "best practice" è di salvare lo stato di verità in un bucket su GCS, a cui punta il codice, in modo tale da, se ci fossero cambiamenti, applicare solamente quest'ultimi senza ricaricare totalmente il progetto "ex - novo".

```
terraform {
    backend "gcs" {
    bucket = "tesietldomenico-terraform"
    prefix = "dev/state"
} }
```

Come descritto nel suddetto capitolo, Terraform è specificatamente progettato per supportare un'architettura software a plugin, dove ciascuna piattaforma, come GCP, AWS o Azure può sviluppare providers (connettori) che permettano di definire risorse rappresentanti i servizi messi a disposizione. Data la sua popolarità, Google Cloud è uno dei provider più supportati: è quindi possibile implementare virtualmente ciascuna risorsa ed API di GCP. Tra le risorse infrastrutturali gestite nel progetto, vi sono tutte le componenti necessarie alla realizzazione dell'ELT alla base della della pipeline in questione, in particolare definiamo due bucket GCS rispettivamente per i file provenienti da Salesforce Marketing Cloud e per i file di output con i risultati della pipeline giornaliera di export.

```
module "import_bucket" {
  source     = "github.com/GoogleCloudPlatform/
                cloud-foundation-fabric.git?
                ref=v23.0.0/modules/gcs"
  project_id = var.project_id
  name       = "etl-tesi-${var.environment}-data-ingest"
```

```
  versioning = true
  location = "europe-west1"
  storage_class = "REGIONAL"
  uniform_bucket_level_access = false
}

module "output_bucket" {
  source     = "github.com/GoogleCloudPlatform/
                cloud-foundation-fabric.git?
                ref=v23.0.0/modules/gcs"
  project_id = var.project_id
  name       = "etl-tesi-${var.environment}-data-output"
  versioning = true
  location = "europe-west1"
  storage_class = "REGIONAL"
  uniform_bucket_level_access = false
}
```

Si definiscono, inoltre, tutte le componenti necessarie al rilascio del cluster Composer, tra cui la VPC (Virtual Private Cloud) e le subnetwork interne dove comunicheranno le macchine del cluster; si è scelto, per questo particolare progetto, un deployment di tipo pubblico, analogamente a quanto descritto nel precedente capitolo.

```
resource "google_composer_environment" "etl-tesi-composer" {
  provider =  google-beta
  name    = "etl-tesi-composer"
  region  = var.region
  project = var.project_id
  config {
    software_config {
      image_version  = var.composer_image_version
      env_variables = {
        AIRFLOW_VAR_BIGQUERY_LOCATION = var.region
```

```
       }
    }

    workloads_config {
      scheduler {
        cpu        = 0.5
        memory_gb  = 1.875
        storage_gb = 1
        count      = 1
      }
      web_server {
        cpu        = 0.5
        memory_gb  = 1.875
        storage_gb = 1
      }
      worker {
        cpu = 0.5
        memory_gb  = 1.875
        storage_gb = 1
        min_count  = 1
        max_count  = 3
      }
    }
    environment_size = "ENVIRONMENT_SIZE_SMALL"

    node_config {
      network         = google_compute_network.vpc_network.self_link
      subnetwork      = google_compute_subnetwork.composersub.self_link
      service_account = module.composer_sa.email

    }
  }
}
```

5.2 Pipeline di Orchestrazione tramite Airflow

L'infrastruttura portante dei processi analitici moderni è il software orchestrativo che detta i tempi e gestisce le dipendenze tra i vari task. Nel nostro specifico caso progettuale, i task sono i seguenti:

- Caricamento dei file CSV da Google Cloud Storage verso tabelle predisposte in un dataset su BigQuery
- Esecuzione delle query SQL in BigQuery, processanti le trasformazioni sul batch corrente e producenti, in output, tabelle con i risultati da esportare in CSV
- Export effettivo dei dati su file .csv sempre in GCS, da cui il sistema rendicontativo può recuperare i suddetti

Come nel progetto originario, supponiamo che Salesforce Marketing Cloud abbia la capacità di esportare i file giornalieri sul bucket GCS *etl-tesi-dev-data-ingest* nel progetto. Il primo DAG ha il compito di effettuare il caricamento dei dati in delle tabelle specchio su BigQuery, quest'ultime definite, gestite e create tramite Terraform. Siamo in un paradigma ELT, il dato viene caricato grezzo direttamente sul Cloud DW senza effettuare trasformazioni preliminari, in un dataset specificatamente predisposto chiamato *marketing_raw*. Siccome ogni file può essere caricato in maniera indipendente, i task possono essere parallelizzati: questo significa che ogni task, a cui ho dato $gcs\_file.split("/")[-1]$, non ha dipendenze in esecuzione; inoltre, il codice può essere reso succinto creando i vari task in un "for loop", sfruttando la capacità che ha Airflow di interpretare il codice Python e creare i task definiti dal *GCSToBigQueryOperator*. In questo specifico use case, essendo che la fase di "Extraction" comprenda, comunque, una sorgente sotto - prodotto di Google come GCS, l'estrazione viene realizzata abbastanza semplicemente con l'ausilio del suddetto operatore, collassando, alla fin fine, con il "Load" come fosse una singola operazione; ovviamente, il caso generale comprende logiche differenti e mediamente più complesse, essendo che,

spesso, le sorgenti non sono sistemi fortemente integrati con BigQuery come GCS. Il parametro *write_disposition*, settato a $WRITE\_TRUNCATE$, sovrascrive qualsiasi tipo di dato nella tabella di staging.

```
from airflow import DAG
from airflow.providers.google.cloud.transfers.gcs_to_bigquery import GCSToBigQue
from airflow.utils.dates import days_ago
from google.cloud import storage
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"]="/Users/domenicodemarchis/
Desktop/ProgettoPOC/tf-sa.json"

# Define default arguments
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
}

# Function to get the first 10 files from a GCS bucket
def list_gcs_files(bucket_name, prefix=''):
    storage_client = storage.Client()
    bucket = storage_client.get_bucket(bucket_name)
    blobs = bucket.list_blobs(prefix=prefix)
    files = [blob.name for blob in blobs]
    return files


file_to_table_id = {
    "cliente.csv": "cliente",
```

```
    "emailclick.csv": "emailclick",
    "emailinvii.csv": "emailinvii",
    "emailunsub.csv": "emailunsub",
    "journey.csv": "journey",
    "journeyActivity.csv": "journeyActivity",
    "notificheclick.csv": "notificheclick",
    "microesiti.csv": "microesiti",
    "notifiche.csv": "notifiche",
    "sms.csv": "sms"
}

table_to_schema= {
    "cliente.csv": [
            {'name': 'cod_istituto', 'type': 'INTEGER',
            'mode':  'NULLABLE'},
            {'name': 'COD_NDG_ANAGRAFICA_NSG', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'COD_FISCALE_PARTITA_IVA', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'SubscriberKey', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'EMAIL_ADDRESS', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'Numero_Telefono', 'type': 'STRING',
            'mode': 'NULLABLE'}
    ],

    "emailclick.csv": [
            {'name': 'sendid', 'type': 'INTEGER',
            'mode':'NULLABLE'},
            {'name': 'EventDate', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'SubscriberKey', 'type': 'INTEGER',
```

```
        'mode': 'NULLABLE'},
],
"emailinvii.csv": [
        {'name': 'activity_id', 'type': 'INTEGER',
        'mode': 'NULLABLE'},
        {'name': 'sendid', 'type': 'INTEGER',
        'mode': 'NULLABLE'},
        {'name': 'data_invio', 'type': 'STRING',
        'mode': 'NULLABLE'},
        {'name': 'EventDate', 'type': 'STRING',
        'mode': 'NULLABLE'},
        {'name': 'SubscriberKey', 'type': 'INTEGER',
        'mode': 'NULLABLE'},
],
"emailunsub.csv": [
        {'name': 'sendid', 'type': 'INTEGER', 'mode':
        'NULLABLE'},
        {'name': 'EventDate', 'type': 'STRING', 'mode':
        'NULLABLE'},
        {'name': 'SubscriberKey', 'type': 'INTEGER', 'mode':
        'NULLABLE'},
],
"journey.csv": [
        {'name': 'name', 'type': 'STRING',
        'mode': 'NULLABLE'},
        {'name': 'id', 'type': 'INTEGER',
        'mode': 'NULLABLE'},
        {'name': 'JourneyStatus', 'type': 'STRING',
        'mode': 'NULLABLE'},

],
"journeyActivity.csv": [
        {'name': 'id', 'type': 'INTEGER',
```

```
            'mode': 'NULLABLE'},
            {'name': 'JourneyID', 'type':
            'INTEGER', 'mode': 'NULLABLE'},
            {'name': 'Activity_Name', 'type': 'STRING',
            'mode': 'NULLABLE'}
    ],
    "notificheclick.csv": [
            {'name': 'id', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'ClickDate', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'deviceId', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'notificaId', 'type': 'STRING',
            'mode': 'NULLABLE'},

    ],
    "microesiti.csv": [
            {'name': 'ESITOMC', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'MICROESITO', 'type': 'STRING',
            'mode': 'NULLABLE'}
    ],
    "notifiche.csv": [
            {'name': 'id', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'DateTimeSend', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'deviceId', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'testo', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'status', 'type': 'STRING',
```

```
            'mode': 'NULLABLE'},
            {'name': 'SubscriberKey', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'activity_id', 'type': 'INTEGER',
            'mode': 'NULLABLE'},

    ],
    "sms.csv": [
            {'name': 'id', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'logDate', 'type': 'STRING',
            'mode': 'NULLABLE'},
            {'name': 'delivered', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'undelivered', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'SubscriberKey', 'type': 'INTEGER',
            'mode': 'NULLABLE'},
            {'name': 'activity_id', 'type': 'INTEGER',
            'mode': 'NULLABLE'},

    ]
}

# cod_istituto, COD_NDG_ANAGRAFICA_NSG, COD_FISCALE_PARTITA_IVA,
#SubscriberKey, EMAIL_ADDRESS
# Define the DAG
with DAG(
    'gcs_to_bigquery',
    default_args=default_args,
    description='A DAG to move the marketing files from GCS to BigQuery',
    schedule_interval=None,
    start_date=days_ago(1),
```

```
    catchup=False,
) as dag:

    # Define variables
    bucket_name = 'etl-tesi-dev-data-ingest'
    prefix = ''
    dataset_id = 'marketing_raw'
    # table_id = ''

    # Get the list of files to move
    gcs_files = list_gcs_files(bucket_name, prefix)

    # Create a task for each file
    for gcs_file in gcs_files:
        filename = gcs_file.split("/")[-1]
        if filename not in file_to_table_id.keys():
            continue
        table_id = file_to_table_id[filename]
        gcs_to_bq_task = GCSToBigQueryOperator(
            task_id=f'gcs_to_bq_{filename}',
            bucket=bucket_name,
            source_objects=[gcs_file],
            destination_project_dataset_table=f'{dataset_id}.{table_id}',
            schema_fields=table_to_schema[filename],
            autodetect=False,
            skip_leading_rows=1,
            write_disposition='WRITE_TRUNCATE'
        )
```

Il DAG finale che trasferisce i file da BigQuery a Google Cloud Storage è molto simile: per questo use case, il provider Google di Airflow mette a disposizione l'operatore *BigQueryToGCSOperator*, facente internamente utilizzo delle API di export di BigQuery per esportare il file in formato CSV.

```
for bq_table in bq_tables:
    gcs_path = f'gs://{gcs_bucket}/{bq_table}'

    export_task = BigQueryToGCSOperator(
        task_id=f'export_{bq_table}',
        source_project_dataset_table=f"etl-tesi-domenico.
        marketing_final.{bq_table}",
        destination_cloud_storage_uris=[f'{gcs_path}/{bq_table}_*.csv'],
        export_format=export_format,
        print_header=True,
        field_delimiter=','
    )
```

Nel mezzo tra l'import e l'export, la logica principale di orchestrazione è definita nel DAG *bigquery_execute_sql_files*.
Le varie query definenti la logica di business del processo sono racchiuse in file .sql all' interno della repository del progetto e vengono lanciate tramite l'operatore Airflow *BigQueryInsertJobOperator*, di cui riportiamo la query "partition" a titolo di esempio.

```
def openquery(filename):
    with open(f"/Users/domenicodemarchis/Desktop/ProgettoPOC
    /bigquery/sql/{filename}.sql") as f:
        return f.read()


partition = BigQueryInsertJobOperator(
    task_id="partition",
    configuration={
        "query": {
             "query": openquery("1_partition"),
             "useLegacySql": False,
        }
    },
    location="EU",)
```

Nella dichiarazione dei task ho utilizzato una funzioanalità di Airflow che permette di inserire delle macro che modificano il codice prima che quest'ultimo venga interpretato, una sorta di "riflessione" per intenderci: in questo caso, la macro definita tra {% %} include i contenuti del file .sql corrispondente al task dove viene inserita la query. Il DAG nella POC istanzia sette di questi operatori, nel progetto originale con la Banca oltre 80, alcuni di essi dipendenti tra di loro; il codice istanzia dapprima ogni task e, successivamente, gestisce in un'unica riga di codice le dipendenze.

```
partition >> merge >> prep >> [campaign, sms, notifiche, email]
```

In particolare, l'ordinamento topologico del DAG fa si che i task "partition", "merge" e "prep" vengano schedulati uno dopo l'altro e, una volta completato il task prep, possano partire in parallelo i task di campaign, sms, notifiche ed email. Quest'ultimi, infatti, lavorano in lettura sulle tabelle create nei task precedenti e producono, rispettivamente, i 4 file di output obiettivo della POC, 44 nel progetto originario.

5.3 Trasformazioni e Logiche di Business con BigQuery

Nel "Capitolo 3" abbiamo analizzato l'architettura fisica di un Cloud Datawarehouse come BigQuery e il meccanismo alla base dell'efficiente processamento di PetaByte di dati grazie ai cluster Google Cloud. Per la POC, BigQuery è usato come "mero" mezzo di trasformazione del dato: non definisce, quindi, un "Data Warehouse" nella stessa accezione espressa nel "Capitolo 2", nemmeno vi sia uno "Star Schema" come logica di gestione del dato o ottimizzazioni analitiche di alcun tipo. Quest'aspetto è particolarmente interessante, in quanto, sebbene BigQuery sia utilizzato praticamente come Cloud DW ROLAP nella stragrande maggioranza dei casi, è ideale anche come motore per effettuare trasformazioni su grandi volumi di dati ed efficientare pipeline che si interfaccino, come nel nostro caso, a sistemi fortemente legacy. Nel progetto reale, le performance di BigQuery hanno

permesso di ridurre il tempo impiegato per effettuare l'originale pipeline da 3 ore a 15 minuti, ottentendo una riduzione di oltre il 90% rispetto a Salesforce Automation. Ciò di cui discutiamo con la POC non è per nulla distante dal caso analitico, in quanto qui ci si sta collocando nello step direttamente precedente, dove viene fornita la "materia prima" su cui effettuare le suddette computazioni; quest'ultimo caso si differenzierebbe unicamente nella modellazione effettiva dello "Star Schema" tramite la creazione di Fact e Dimension Tables in BigQuery tramite SQL, ma dove, ovviamente, continuino a non esistere computazioni senza una pipeline preliminare di iniezione e pulizia del dato. Il layer di trasformazione è organizzato in 4 dataset BigQuery differenti: "marketing_raw", "marketing_source", "marketing_prep" e "marketing_final", rappresentanti tutti i diversi stadi della pipeline; bisogna pensare ad un dataset BigQuery come ad un contenitore di tabelle, viste, routine... simile ad uno "schema" in un DB come PostgreSQL. Il dato grezzo viene importato in marketing_raw attraverso il primo DAG di import: le tabelle al suo interno vengono ri-sovrascritte totalmente ad ogni run del DAG, in quanto "Marketing Cloud" produce i .csv quotidianamente.
I file sorgente rappresentano campagne di marketing promosse dalla Banca ed inviate tramite canali di comunicazione diversi: email, sms e notifiche sul cellulare. I file "sms.csv", "notifiche.csv" e "notificheclick.csv", "emailinvii.csv", "emailunsub.csv" ed "emailclick.csv" racchiudono informazioni relative a questi canali indipendenti. La tabella "cliente.csv" contiene tutta l'anagrafica cliente della "Banca", con informazioni personali e riservate come email, nome e cognome, partita iva... i restanti "journey.csv", "journeyActivity.csv" e "microesiti.csv" contengono, rispettivamente, informazioni sulle campagne marketing, sulle singole attività delle stesse e, in ultima, dei particolari "label" funzionali a classificare le singole entità, in base al che, ad esempio, l'email sia di "unsubscription" o l'sms "undelivered"... necessari alla corretta logica di funzionamento del rendicontativo. Il primo step nella fase di trasformazione è effettuare il parsing delle Date dalle varie tabelle: BigQuery ha, ovviamente, un datatype "DATE" per gestire timestamp di questo tipo e tutte le pecularità caratterizzanti le operazioni che li comprendano, cui anni bisestili, addizione e sottrazione di anni, mesi, giorni... Il tipo

DATE di BigQuery non è nativo dei CSV prodotti da Marketing Cloud, anzi, i timestamp vengono realizzati tramite stringhe comprensive, spesso, di formattazioni molto diverse da quella con cui lavori il rendicontativo, contenendo informazioni irrilevanti come ore, minuti o millisecondi; le trasformiamo come segue, prendendo a titolo di esempio la tabella specchio di "sms.csv" in BigQuery, tutte le altre seguono la stessa esatta logica:

```
CREATE OR REPLACE TABLE marketing_source.sms_source as
SELECT *,
PARSE_DATE("%Y-%m-%d", SPLIT(logDate,' '))[OFFSET(0)] as logDateDate
FROM marketing_raw.sms;
```

Ho gestito questi timestamp utilizzando la funzione SPLIT(), producente un Array BigQuery di cui ho preso solamente la prima parse: in questo modo si risolve il problema delle informazioni irrilevanti escludendo ore, minuti e millisecondi; il tutto insieme alla funzione PARSE_DATE per effettuare, appunto, la conversione nel suddetto tipo DATA. Oltre i motivi precedentemente esposti, la conversione a DATA è strettamente necessaria alla logica di business in quanto si esige di produrre gli ultimi 3 giorni per ogni file di output, indipendentemente dal che Marketing Cloud produca CSV giornalieri, nonché debba essere possibile gestire recuperi su un range temporale di 6 mesi, il che si ricollega con i discorsi sulla "profondità temporale" del dato discussi nel "Capitolo 2", lavorando analogamente i DW; per gestire questa retention, si è creato nel dataset marketing_source un'altro dataset con tabelle partizionate sulla Data. Il secondo step della pipeline in 2_*merge*, difatti, effettua un merge dei dati di input dalle varie tabelle processate nello step precedente verso le tabelle *_all* create direttamente all'interno di BigQuery e posizionate sempre in "marketing_source", funzionali, proprio, a mantenere il suddetto storico; prendiamo ad esempio l'esecuzione sugli SMS che segue.

```
MERGE marketing_source.sms_all a
USING marketing_source.sms_source t
ON a.id = t.id
WHEN NOT MATCHED THEN INSERT ROW;
```

```
CREATE OR REPLACE TABLE marketing_source.sms as
SELECT * FROM marketing_source.sms_all t
WHERE t.logDateDate > current_date - 3 ;
```

La funzione di "MERGE" permette di mantenere i vincoli sulle tuple duplicate, classici di un DB relazionale, aggiungendo ad "sms_all" solo i nuovi record prodotti giornalmente, per poi partizionare la precedente tabella in una nuova "sms" che filtra lo storico degli ultimi 3 giorni necessario in output; le logiche sono esattamente le stesse per tutte le restanti entità.
Da qui si procede alla fase successiva di "prep", in cui effettuiamo le ultime trasformazioni necessarie all'esportazione dei risultati in output. Una delle problematiche che risolviamo in "prep" è relativa ad un conflitto sui timestamp di Salesforce, aventi un fuso orario di 7 ore diverso da quello Italiano. Si è reso necessario, quindi, interpretare i timestamp utilizzando il sistema di timezone BigQuery in America/Chicago come punto di riferimento e convertendo il timestamp in Europe/Rome; riportiamo un esempio sempre in relazione agli SMS, logiche del tutto analoghe sono adottate per convertire anche i restanti canali.

```
CREATE OR REPLACE TABLE marketing_prep.sms
AS SELECT
s.*,
FORMAT_DATETIME(
"%Y-%m-%d %H:%M:%S",
DATETIME(TIMESTAMP(PARSE_DATETIME("%Y-%m-%d %H:%M:%S", s.logDate),
'America/Chicago'), 'Europe/Rome'))
AS ymd_hms_logDate,
FROM marketing_prep.sms s
WHERE length(cast(s.logDate as string)) <= 19;
```

Riguardo quest'ultimo aspetto c'è da fare leggermente chiarezza, in quanto sembri in antitesi con ciò che abbiamo discusso precedentemente: perché il

"FORMAT_DATETIME" cerca di formattare il timestamp temporale tenendo conto di ore, minuti e secondi se, precedentemente, le avevamo "bollate" come irrilevanti? In realtà, il rendicontativo necessita di CSV in output che siano provvisti delle suddette informazioni nei propri timestamp e l'irrilevanza di quest'ultime è relativa alla logica con cui stiamo costruendo l'output: essendo richiesti gli ultimi 3 giorni, sfruttiamo il tipo "DATE" di BigQuery sia per costruire lo storico *_all* che per filtrarlo, sebbene in uscita riporteremo, più semplicemente, i timestamp origine opportunamente convertiti. In questa fase filtriamo, inoltre, tutte le campagne di marketing non attive, in quanto i file da produrre in output necessitano di una classificazione delle email, notifiche ed sms che sono strettamente relativi a campagne in corso.

```
CREATE OR REPLACE TABLE marketing_prep.JourneyPrep
as
SELECT
j.*,
ja.id as activity_id,
ja.Activity_Name
FROM marketing_raw.journey j
INNER JOIN marketing_raw.journeyActivity ja ON j.id = ja.JourneyID
WHERE j.JourneyStatus='Running';
```

Può capitare, inoltre, che nelle campagne SMS ci siano dei casi in cui le colonne delivered e undelivered siano valorizzate entrambe ad 1; questo può significare un retry dell'invio del messaggio e il sistema rendicontantivo richiede la produzione di due record. In questo caso, si è usato un semplice UNION ALL per gestire la richiesta come segue.

```
CREATE OR REPLACE TABLE marketing_prep.sms
AS SELECT *,
FROM marketing_source.sms p
WHERE delivered=1
UNION ALL
SELECT *,
```

```
FROM marketing_source.sms p
WHERE undelivered=1
```

Avendo fatto assiduamente riferimento al canale SMS per spiegare tutte le logiche implementative, riportiamo per completezza una possibile configurazione del file di output "esitisms" e la query generante, forti del che molto dell'evincibile dalla seguente rimanga, anche qui, diametralmente analogo per i restanti canali; si rimanda, comunque, al codice completo nella repository GitHub per una più dettagliata consultazione, dove sono riportati anche i CSV sorgente a cui ho fatto riferimento per la costruzione della pipeline, il link è reperibile in "Appendice" dove discutiamo, inoltre, anche di un breve approfondimento sui risultati applicativi della POC.

```
CREATE OR REPLACE TABLE marketing_final.esitisms AS
SELECT
c.cod_istituto AS COD_ABI, -- codice istituto bancario
c.COD_NDG_ANAGRAFICA_NSG AS COD_NDG, -- codice anagrafica
c.COD_FISCALE_PARTITA_IVA AS COD_FISCALE,
c.EMAIL_ADDRESS AS EMAIL,
jp.Activity_Name AS ACTIVITY_NAME,
s.ymd_hms_logDate  AS DATA_INVIO,
m.MICROESITO AS COD_MICROESITO,
"SMS" AS COD_CANALE
FROM marketing_prep.sms s
INNER JOIN marketing_raw.cliente c
            ON s.SubscriberKey = c.SubscriberKey
INNER JOIN marketing_prep.JourneyPrep jp
            ON s.activity_id = jp.activity_id
INNER JOIN marketing_raw.microesiti m
        ON CASE
            WHEN s.Undelivered = 1 THEN 'SMS_Undelivered'
            WHEN s.Delivered = 1 THEN 'SMS_Delivered'
            ELSE 'SKIPPARE'
            END = m.ESITOMC
```

Il concetto alla base è, proprio, costruire dei CSV singoli in cui convergano le informazioni più sensibili da ciascun file sorgente, classificando il canale di comunicazione con l'opportuno microesito, collegandolo al proprio timestamp, cliente coinvolto e attività di marketing relativa; si noti come nel CSV di output, come suddetto, il timestamp temporale sia una conversione di quello in input e non il DATE type. Le restanti query di "esito" creano, all'effettivo, le rimanenti tabelle di output in "marketing_final", dataset a cui fa riferimento il DAG Airflow di exporting discusso già alla sezione precedente.

COD_ABI	COD_NDG	COD_FISCALE	EMAIL	ACTIVITY_NAME	DATA_INVIO	COD_MICROESITO	COD_CANALE
123456	NDG001	IT12345678901	user1@example.com	Campaign1_SocialADV	06/07/2024 17:20	SMS_DEL	SMS
123456	NDG001	IT12345678901	user1@example.com	Campaign1_Survey	06/07/2024 17:15	SMS_DEL	SMS
345678	NDG003	IT12345678903	user3@example.com	Campaign3_Event	06/07/2024 17:16	SMS_DEL	SMS
345678	NDG003	IT12345678903	user3@example.com	Campaign3_PaidAds	06/07/2024 17:21	SMS_DEL	SMS
567890	NDG005	IT12345678905	user5@example.com	Campaign5_Referral	07/07/2024 17:17	SMS_DEL	SMS
567890	NDG005	IT12345678905	user5@example.com	Campaign5_Referral	07/07/2024 17:22	SMS_DEL	SMS
789012	NDG007	IT12345678907	user7@example.com	Campaign7_WebsiteUpdate	07/07/2024 17:23	SMS_DEL	SMS
789012	NDG007	IT12345678907	user7@example.com	Campaign7_CustomerSurvey	07/07/2024 17:18	SMS_DEL	SMS
234567	NDG009	IT12345678909	user9@example.com	Campaign9_LoyaltyProgram	06/07/2024 17:19	SMS_DEL	SMS
234567	NDG009	IT12345678909	user9@example.com	Campaign9_PressRelease	06/07/2024 17:24	SMS_DEL	SMS
234567	NDG002	IT12345678902	user2@example.com	Campaign2_Webinar	06/07/2024 17:21	SMS_UND	SMS
234567	NDG002	IT12345678902	user2@example.com	Campaign2_Welcome	08/07/2024 17:16	SMS_UND	SMS
456789	NDG004	IT12345678904	user4@example.com	Campaign4_AnalyticsReview	07/07/2024 17:22	SMS_UND	SMS
456789	NDG004	IT12345678904	user4@example.com	Campaign4_SocialADV	07/07/2024 17:17	SMS_UND	SMS
678901	NDG006	IT12345678906	user6@example.com	Campaign6_Promotion	07/07/2024 17:18	SMS_UND	SMS
678901	NDG006	IT12345678906	user6@example.com	Campaign6_DirectMail	07/07/2024 17:23	SMS_UND	SMS
123456	NDG008	IT12345678908	user8@example.com	Campaign8_FreeTrial	07/07/2024 17:24	SMS_UND	SMS
123456	NDG008	IT12345678908	user8@example.com	Campaign8_FreeTrial	08/07/2024 17:19	SMS_UND	SMS
123456	NDG010	IT12345678910	user10@example.com	Campaign10_SocialMediaChallenge	08/07/2024 17:25	SMS_UND	SMS
123456	NDG010	IT12345678910	user10@example.com	Campaign10_Promotion	06/07/2024 17:20	SMS_UND	SMS

Figura 5.1: Output File - esitisms.csv

6

Considerazioni Finali

In quest'elaborato di tesi si sono trattati, nella loro generalità, molti dei concetti e dei pattern moderni più in voga nel Big Data Management. Come già assiduamente trattato, una data governance efficiente si dimostra una qualità critica in, praticamente, qualunque aspetto della vita di un'impresa, tanto che risulti di fondamentale importanza costruire, ad oggi, soluzioni che nel futuro possano non dimostrarsi best practice, ma che siano facilmente estendibili a quelle che saranno esigenze e sviluppi futuri; per chiarire meglio quest'aspetto, lo use case della POC precedentemente discussa è alquanto emblematico: il software rendicontativo della Banca è uno tra i sistemi che quest'ultima utilizza per, come suddetto, realizzare i processi di BI a valle e, sicuramente, non si dimostra la miglior soluzione possibilmente percorribile, essendo che molte delle computazioni effettuate potrebbero essere realizzate direttamente in BigQuery e, d'altronde, con una POC molto simile a quella realizzata. La domanda che sorge spontanea, quindi, è perché si continui a fare affidamento a software, duplice la rapidità con cui si sia evoluto il mercato IT negli ultimi anni, pressoché obsoleti e incapaci di soddisfare le esigenze

moderne; la risposta coincide, proprio, con quest'ultimo aspetto: l'evoluzione di Internet, i Big Data, le architetture distribuite, il Cloud... costituiscono un parco di soluzioni che, se opportunamente utilizzate, permettono di ovviare alla necessità di grande capacità computazionale e analitiche real time, sebbene quest'ultime, come discusso nei primissimi capitoli, siano necessità molto diverse da ciò che richiedesse il mercato a cavallo tra la fine degli anni '90 e l'inizio del 2000. Progettare sistemi informatici moderni vuol dire soprattutto costruire soluzioni che possano interfacciarsi facilmente con strumenti futuri e ovviare alle necessità ed esigenze che saranno, adattandovisi; proprio in questo senso, il concetto di "scalabilità", fondamentale per le nostre discussioni, viene esteso non solamente alle risorse prettamente fisiche del sistema, ma all'architettura essa stessa, intesa come la capacità di quest'ultima ad integrarsi con sistemi esterni. Nelle soluzioni moderne di Data Management, avevamo discusso come tra le caratteristiche fondamentali nella progettazione di pattern ETL ed ELT ci fosse, proprio, questo concetto di integrazione, estendibile poi al caso generalissimo; la POC progettata permette di efficientare il processo complessivo di BI da parte della Banca, sebbene si limiti unicamente ad eliminare l'importante "collo di bottiglia" che ci fosse nella fase preliminare di pulizia ed iniezione del dato, non ad incrementare l'aspetto computazionale esso stesso: proprio in questo senso, tale software rendicontativo non rispetta gli standard moderni di scalabilità, o meglio, integrazione precedentemente discussi, e destinato, quindi, ad una completa sostituzione nel breve periodo, come con buona approssimazione sarà. Come accennato precedentemente, la POC progettata risulta facilmente estendibile al caso in cui si decida di esularsi dal software rendicontativo e concentrare anche la fase computazionale in BigQuery, difatti il concetto stesso alla base del lavoro di consulting come Accenture è fornire soluzioni efficienti nel breve e nel lungo periodo, adattandosi agilmente alla risoluzione di problemi già noti in partnership precedenti col cliente, creando una vera e propria collaborazione possibile solo se, appunto, si abbia tassativamente chiaro questo concetto di "integrazione" nei prodotti forniti.

Appendice

La repository GitHub contenente il codice utilizzato per lo sviluppo del progetto "POC: Pipeline degli esiti di campagne di Marketing per una Banca" è consultabile al seguente link di riferimento:
"https://github.com/ddemarchis11/POCtirocinio".

Riportiamo, adesso, alcuni dei risultati relativi all'implementazione effettiva del progetto; come accennato precedentemente, la POC ha avuto un'importante risonanza in termini di dimunuzione del tempo necessario all'esecuzione dei batch, passando da un iniziale media di 3 ore a, circa, 15 minuti, con una riduzione approssimativamente del 90% sul tempo iniziale. L'aspetto di fondamentale importanza da sottolineare è che, ovviamente, la POC non dispone a priori dell'effettiva dimensione dei record forniti da Marketing Cloud, conseguentemente tutti i valori a cui faremo riferimento sono stime e medie riguardanti l'esecuzione effettiva della POC in più giorni consecutivi, pesandoli opportunamente in funzione, proprio, della dimensione dei batch.

	Tempo Precedente POC	Tempo Nuova POC	Delta	Delta Throughput
Giorno 1	155	85	−70	+1781
Giorno 2	141	81	−60	+1008
Giorno 3	146	78	−68	+1383
Giorno 4	152	83	−69	+1599
Giorno 5	162	91	−71	+1649
Giorno 6	142	85	−57	+1538

Come suddetto, si sono valutate diverse esecuzioni nell'arco di 20 giorni lavorativi, fornendo un parco di informazioni che permetta di approssimare abbastanza bene l'andamento prestazionale generalissimo della POC; ne sono riportati solo i primi 6, abbastanza per non complicare eccessivamente l'analisi. Contestualizzando i dati riportati in tabella, dai valori di "*Delta*" è possibile corroborare quanto discutessimo: per "*Delta*" intendiamo la differenza tra il tempo impiegato per l'esecuzione della nuova POC e quella precedente, realizzandone una media che li pesi funzionalmente alla dimensione dei batch relativi, non riportate poiché poco funzionali al discorso, si ottiene un valore di ≈ 70, corrispondente ad un'attenuazione di $\approx 50\%$ sul tempo iniziale; sebbene quest'ultimo aspetto potrebbe stridere con quanto discusso precedentemente, in realtà i dati riportati sono relativi ad una valutazione fatta a Febbraio per una primissima implementazione della POC, il prodotto finale effettivamente consegnato al cliente, verso l'inizio di Aprile, presentò significative parallelizzazioni delle task, come già ampiamente trattato nel "Capitolo 5", da cui segue l'ulteriore incremento prestazionale. In tabella sono riportate anche delle voci "Delta Throughput": quest'ultime sono indicanti del numero di record processati in più al secondo dalla nuova POC, particolare non intaccato dalla parallelizzazione, potendo apprezzare anche un'ottimizzazione sotto quest'ultimo aspetto.

Bibliografia

[1] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, and R. Torlone, *Basi di Dati*. Milano: McGraw-Hill, 2023.

[2] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 215–226, 2016.

[3] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min, *et al.*, "Dremel: A decade of interactive sql analysis at web scale," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3461–3472, 2020.

[4] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, vol. 8, p. 28, 2021.

[5] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill Education, 2020.

[6] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The definitive guide to dimensional modeling*. Wiley, 2013.

[7] K. L. Jackson and S. Goessling, *Architecting Cloud Computing Solutions: Build Cloud Strategies that align technology and economics while effectively managing risk*. Packt Publishing, 2018.

[8] D. Sullivan, *Official Google Cloud Certified Associate Cloud Engineer Study Guide*. Sybex, a Wiley brand, 2019.

[9] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, 2015.

[10] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1107–1117, 2013.

[11] V. Rainardi, *Building a data warehouse*. Apress, 2014.

[12] B. Devlin, *Data Warehouse: From architecture to implementation*. Addison-Wesley, 2000.

[13] A. Widjaja, *Data Engineering with Google Cloud Platform: A practical guide to operationalizing Scalable Data Analytics systems on GCP*. Packt Publishing, 2022.

[14] D. Laney, "3D data management: Controlling data volume, velocity, and variety," tech. rep., META Group, February 2001.

[15] B. K. Pandey and E. R. Schoof, *Building ETL pipelines with python create and deploy enterprise-ready ETL pipelines by employing modern methods*. Packt Publishing, Limited, 2023.

[16] M. H. Ali, M. S. Hosain, and M. A. Hossain, "Big data analysis using bigquery on cloud computing platform," *Australian Journal of Engineering and Innovative Technology*, 2021.

[17] W. H. Inmon, *Building the data warehouse*. Wiley, 2011.

[18] V. Lakshmanan and J. Tigani, *Google Bigquery: The definitive guide: Data warehousing, analytics, and machine learning at scale*. O'Reilly, 2020.

[19] Google, "BigQuery Official Documentation." https://cloud.google.com/bigquery/docs?hl=it.

[20] Apache, "Airflow Official Site." https://airflow.apache.org/.

[21] Apache, "Airflow Official Documentation." https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html.

[22] Apache, "Airflow Documentation UI." https://airflow.apache.org/docs/apache-airflow/stable/ui.html.

[23] Redis, "Redis NoSQL Documentation." https://redis.io/nosql/what-is-nosql/.

[24] Altexsoft, "ELT vs ETL: Key Differences Everyone Must Know." https://www.altexsoft.com/blog/etl-vs-elt.