POLITECNICO DI MILANO
Corso di Laurea Magistrale in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Progettazione e realizzazione di una procedura di integrazione per dati genomici

Relatore: Alessandro Campi
Correlatore: Anna Bernasconi

Tesi di Laurea di:
Federico Gatti, matricola 852377

Anno Accademico 2016-2017

A tutte le persone che mi hanno sostenuto

Indice

Sommario **IX**

Abstract **XI**

Ringraziamenti **XIII**

1 Introduzione **1**
1.1 Situazione attuale 1
1.2 Lavoro proposto 2
1.3 Struttura della tesi 3

2 Stato dell'arte **5**
2.1 Il sequenziamento del DNA: la Next Generation Sequencing 5
2.1.1 Preparazione e immobilizzazione del DNA o Creazione della sequencing Library 6
2.1.2 Reazione di amplificazione 7
2.1.3 Reazione di sequenziamento 7
2.1.4 Risultati 7
2.2 Genomic Data Model 8
2.3 GenoMetric Query Language 10
2.4 Sorgenti di dati: Encode e TCGA (GDC) 12
2.5 Extract, transform, load (ETL) 13
2.5.1 Extract 13
2.5.2 Transform 14
2.5.3 Load 14
2.6 Sistemi di integrazione di dati biologici già esistenti 15

3 Integrazione dei dati genomici **17**
3.1 Genomic Conceptual Model 17
3.2 Design 17
3.2.1 Tassonomia degli attributi 18
3.2.2 Source Analysis 19
3.2.3 Altre proprietà 19
3.2.4 Regole 19

3.3 Modello ER . . . 20
3.3.1 Modello Ideale . . . 20
3.3.2 Verifica delle sorgenti sullo schema GCM . . . 24

4 Implementazione del modello concettuale 27
4.1 Definizione di chiave univoche per le tabelle . . . 27
4.2 Ridefinizione del modello . . . 28
4.3 Variazione delle proprietà degli attributi . . . 30
4.4 Revisione delle regole e dipendenze . . . 33
4.5 Introduzione di funzioni per la trasformazione dei dati in input . . . 35

5 Architettura 39
5.1 Utilizzo del linguaggio Scala . . . 39
5.2 Libreria Slick . . . 39
5.3 Struttura generale del programma . . . 40
5.3.1 Indipendenza dei processi ETL . . . 41
5.4 Utilizzo di tool preesistenti . . . 42
5.5 Architettura del codice e delle classi . . . 43
5.5.1 Gestore del Database . . . 43
5.5.2 Model delle tabelle . . . 44
5.5.3 Raccoglitore di tabelle . . . 46

6 Funzionamento dell'applicativo 49
6.1 Sorgenti supportate . . . 49
6.2 Metodi di esecuzione . . . 49
6.3 Tecnica di Inserimento e Update di una tupla . . . 50
6.4 Parametri in Input . . . 50
6.4.1 File dei dati . . . 51
6.4.2 File di configurazione . . . 54
6.4.3 XMLReader . . . 59
6.4.4 Sequenza di operazioni . . . 59
6.5 Schema logico globale del framework . . . 61
6.6 Implementazione delle relazioni e dipendenze fra tabelle . . . 61
6.7 Molteplici Item, Replicati e Donatori per ogni file in Encode . . . 64
6.7.1 Ricerca degli attributi Platform e Pipeline . . . 64
6.8 Replicati biologici e Replicati tecnici . . . 67
6.9 Statistiche e messaggi loggati . . . 69
6.9.1 SLF4J e log4j . . . 73
6.10 Esecuzione del programma in modalità *export* . . . 74
6.10.1 Estrazione dei dati dalle tabelle . . . 75
6.10.2 Possibili messaggi di log . . . 76
6.11 Application Configuration file . . . 78
6.12 Tabella per il supporto alle ontologie . . . 80
6.13 Complessità temporale . . . 81

6.14 Tempi di esecuzione e performance 82
6.14.1 Performance sull'inserimento dei dati 83
6.14.2 Differenza delle operazioni con un database popolato 84
6.14.3 Differenza fra inserimento e update 85
6.14.4 Esecuzioni su macchine differenti 86

7 Estendibilità del framework 87
7.1 Estendibilità nelle funzioni di inserimento e regole di mapping 87
7.2 Introduzione di un nuova sorgente 88
7.3 Altri formati di file in input 88
7.4 Cambio della tecnologia del database 88
7.5 Nuova tabella o attributo 89
7.6 Introduzione di una nuova statistica 90

8 Sviluppi futuri e conclusioni 91

Bibliografia 93

A Tabelle di mapping per il repository Encode 97

B Tabelle di mapping per il repository TCGA 101

C Schema xsm del file di configurazione 105

D Confronto proprietà attributi modello ideale e modello reale 107

E Statistiche delle esecuzioni 111

List of Figures

2.1 Rappresentazione schematica di una sequencing library per la next generation sequencing. a: supporto; b: adattatori; c: frammenti di DNA ancorati al supporto (immobilizzati) tramite gli adattatori. 8
2.2 Tabella delle regioni. 10
2.3 Tabella dei metadati. 11
2.4 Architettura di un sistema ETL 15

3.1 Diagramma ER del GCM 21
3.2 Confronto tra il GCM e lo schema delle sorgenti. 26

4.1 Diagramma ER del GCM modificato: sono indicate con in colore giallo le modifiche e in colore verde le aggiunte 29
4.2 Encode Experiment con le etichette gialle vengono indicati i file, con le etichette azzurre vengono indicate le tipologie di operazioni attuate per la loro elaborazione in un nuovo file processato. 32

5.1 Rappresentazione ad alto livello del framework. 41
5.2 Estratto di diagramma UML Model delle tabelle. 46
5.3 Diagramma UML Raccoglitore delle tabelle. 47

6.1 Schema dei parametri in input 51
6.2 Rappresentazione grafica della struttura del file di configurazione. 55
6.3 Rappresentazioni di possibili operazioni 60
6.4 Dipendenze delle tabelle nella procedura di export 60
6.5 Rappresentazione del grafo aciclico orientato. 68
6.6 Esempio di relazione fra replicati biologici e tecnici. 69
6.7 Dipendenze delle tabelle nella procedura di export 76
6.8 Rappresentazione di un DAG con massimo numero di archi. 82

List of Tables

3.1 Tassonomia degli attributi e valori ortogonali 18

4.1 Chiave univoche per tabella 28
4.2 Elenco delle proprietà degli attributi 31
4.3 Regole di dipendenza intratabellari 34
4.4 Regole di obbligatorietà intratabellari 34
4.5 Regole di dipendenza di Valore. 34
4.6 Regole utilizzate per la verifica di proprietà nel contesto di ENCODE. 35
4.7 Panoramica attuale dello sviluppo delle regole di *Constestualità* e *Dipendenza* nel framework 35
4.8 Funzione per repository ENCODE 36
4.9 Funzioni di trasformazione GDC 37

6.1 Statistiche di riepilogo dello stato dei file per la sorgente ENCODE. 54
6.2 Metodi disponibili di inserimento degli attributi 57
6.3 Esempi di metodi disponibili per l'inserimento degli attributi: il simbolo ✓ indica che per il metodo ha senso definire l'attributo, mentre la $\times$ indica che se anche l'attributo venisse indicato questo non produrrebbe nessun effetto alla procedura di mapping 57
6.4 Statistiche di riepilogo di numero di item inseriti o aggiornati per il repository ENCODE. 65
6.5 Tabella delle ontologie 80
6.6 Riepilogo del numero di inserimenti/aggiornamenti per file in ENCODE. 82
6.7 Tabella riassuntiva per inserimenti dei dati in ENCODE. Le ultime due righe dell'ultima colonna indicano media e deviazione standard. 84
6.8 Tabella riassuntiva per inserimenti dei dati in TCGA. Le ultime due righe dell'ultima colonna indicano media e deviazione standard. 85
6.9 Tabella riassuntiva per inserimenti dei dati in ENCODE con opzione di file derivati e ontologie. Le ultime due righe dell'ultima colonna indicano media e deviazione standard. 85
6.10 Statistiche dell'inserimento del dataset *masked_copy_number_segment* GRCh38 di TCGA nel contesto di database vuoto e con dati. 85
6.11 Statistiche dell'inserimento e aggiornamento del dataset *masked_copy_number_segment* GRCh38 di TCGA. 86

6.12 Statistiche dell'inserimento del dataset *methylation_beta_value* GRCh38 di TCGA su macchine diverse a confronto. 86

A.1 Mapping tabella Donor per repository ENCODE 97
A.2 Mapping tabella BioSample per repository ENCODE 98
A.3 Mapping tabella Replicate per repository ENCODE 98
A.4 Mapping della tabella Item per repository ENCODE 98
A.5 Mapping della tabella Container per repository ENCODE 99
A.6 Mapping della tabella ExperimentType per repository ENCODE 99
A.7 Mapping della tabella Case per repository ENCODE 99
A.8 Mapping della tabella Project per repository ENCODE 99

B.1 Mapping della tabella Donor per repository TCGA 101
B.2 Mapping della tabella Biosample per repository TCGA 102
B.3 Mapping della tabella Biosample per repository TCGA 102
B.4 Mapping della tabella Item per il repository TCGA 102
B.5 Mapping della tabella Container per repository TCGA 103
B.6 Mapping della tabella ExperimentType per repository TCGA 103
B.7 Mapping della tabella Case per repository TCGA 103
B.8 Mapping della tabella Project per repository TCGA 103

D.1 Proprietà attributi entità Item 107
D.2 Proprietà attributi entità Donor 108
D.3 Proprietà attributi entità Biosample 108
D.4 Proprietà attributi entità Replicate 108
D.5 Proprietà attributi entità ExperimentType 108
D.6 Proprietà attributi entità Container 109
D.7 Proprietà attributi entità Case 109
D.8 Proprietà attributi entità Project 109

E.1 Statistiche per il dataset HG19 di ENCODE con 3 campioni. 111
E.2 Statistiche per il dataset HG19 di ENCODE con 3 campioni. 111
E.3 Statistiche per il dataset GRCh38 di ENCODE con 3 campioni. 111
E.4 Statistiche per il dataset GRCh38 di ENCODE con 3 campioni. 111
E.5 Statistiche per il dataset copy_number_segment di TCGA con 4 campioni. . . . 112
E.6 Statistiche per il dataset isoform_expression_quantification di TCGA con 3 campioni. 112
E.7 Statistiche per il dataset mirna_expression_quantification di TCGA con 3 campioni. 112
E.8 Statistiche per il dataset gene_expression_quantification di TCGA con 3 campioni. 112
E.9 Statistiche per il dataset masked_copy_number_segment di TCGA con 3 campioni. 112
E.10 Statistiche per il dataset masked_somatic_mutation di TCGA con 3 campioni. . 112
E.11 Statistiche per il dataset masked_copy_number_segment di TCGA con 3 campioni. 113

Sommario

Molti repository di Open Data di dati genomici, raccolti da consorzi sul web, forniscono dati molto utili per ricerche biologiche o mediche. Questa disponibilità, unita all'obbligo di depositare in repository pubblici tutti i dataset sperimentali usati per dimostrare i risultati di pubblicazioni scientifiche sui genomi, ha permesso di avere a disposizione un ricchissimo insieme di dati pubblici legati alla genomica.
In particolare, questi dataset sono fonti preziose sfruttate dai biologi per validare o arricchire i propri esperimenti: il loro contenuto è documentato da metadata, nient'altro che delle informazioni che descrivono il contesto dell'esperimento.
Tuttavia, l'enfasi associata al *data sharing* non è corrisposta dall'accuratezza della documentazione dei dati: infatti, i metadati non sono standardizzati tra le diverse sorgenti e spesso sono incompleti o non strutturati.
Da questa constatazione ha origine questo lavoro, svoltosi con l'obiettivo di sviluppare, partendo da documentazioni teoriche che si prefiggono la risoluzione dei problemi precedentemente elencati attraverso la definizione di modelli concettuali, un database che collezioni tutti i dati dei maggiori repository presenti in rete e che proponga un unico standard per la rappresentazione dei metadati genomici.
Partendo da un modello concettuale, è stato implementato un framework per l'integrazione dei dati genomici facilmente estendibile per futuri repository di interesse e inseribile nel contesto di sviluppo architetturale del progetto Genomic Computic (GeCo).
In questa tesi, si intende presentare l'architettura del framework, con particolare enfasi sulla versatilità ed estendibilità, e illustrate alcune tra le principali applicazioni sui repository ENCODE e TCGA.

Abstract

Worldwide consortia collect open genomic data and publish them on the web: these repositories are important enablers of biological research. Moreover, all the datasets produced in the contest of publications in genomics must be deposited to public repositories and made available for the research community in order to allow biologists to use these data to validate or to enrich their experiments. The content of these experiments is described by metadata, sets of information explaining the context of the experiment. Even if the quantity of data on the web is increasing, the biggest obstacle to the full exploitation of these data is the lack of standardized metadata structures: sources often don't provide a conceptual design for metadata or, when a structure is available, it is uselessly complex.
This work, starting from a predefined conceptual model, has the aim to design and implement a framework for the integration of genomic metadata extracted from heterogeneous sources on the web. This framework is easily extensible and it allows the integration of future repositories or new features. In this thesis, the conceptual model is described, then the framework architecture is fully described, and, finally, we demonstrate the usage of the framework in to real world scenarios: the integration in our schema of ENCODE and TCGA.

Ringraziamenti

A conclusione di questo lavoro di tesi è doveroso porre i miei più sentiti ringraziamenti alle persone che maggiormente hanno contribuito alla mia realizzazione.

Colgo l'occasione per ringraziare il Prof. Alessandro Campi, relatore, e la Dott.ssa Anna Bernasconi, correlatore: la loro presenza e i loro preziosi consigli si sono rivelati una fonte essenziale a cui attingere per la realizzazione di questo lavoro e la loro disponibilità, scaturita dall'immensa passione che nutrono nel proprio lavoro, mi hanno offerto una concreta possibilità di crescita professionale.

Un ringraziamento particolare ad Alberto e Riccardo, amici e colleghi dalle scuole superiori, con cui ho condiviso questi cinque anni di università, e tutti gli amici di Svoltastudenti con i quali ho animato la rappresentanza studentesca: attraverso questi momenti di collaborazione e di svago, ho avuto l'opportunità di vivere a pieno i valori della vita universitaria e di apprezzare ciò che l'università offre al di là degli studi didattici, arricchendo il mio bagaglio culturale con esperienze che hanno determinato una significativa crescita personale.

Ringrazio Michela per avermi sempre sostenuto ed aiutato, con infinita pazienza, in questi due anni di laurea Magistrale. Grazie per essere sempre stata al mio fianco e per essere qui, anche oggi, in questo giorno importante, a festeggiare con me questo traguardo.

Un ultimo caloroso "grazie" è per la mia famiglia, ed in particolare i miei genitori, per avermi sempre sostenuto e permesso di seguire le mie passioni: la loro costante presenza e i loro consigli mi hanno reso la persona che sono e vorrei che questo obiettivo raggiunto sia, per loro, motivo di soddisfazione e di orgoglio.

Federico,
Milano, 19 aprile 2018.

Capitolo 1

Introduzione

Grazie alle tecniche di *Next Generation Sequencing*, un recente successo dell'ingegneria genetica che permette di leggere la sequenza di DNA su campioni biologici, un grandissimo numero di dataset genomici è divenuto accessibile. Le macchine sequenziatrici effettuano l'analisi primaria dei dati (tipicamente chiamata *primary data analysis*) e producono dataset contenenti dati non processati (di tipo *raw*) le cui dimensioni raggiungono i centinaia di gigabyte (si parla di circa 200 GB per ogni singolo genoma umano). Successivamente, i dati *raw* vengono elaborati durante l'analisi secondaria dei dati [25] (detta *secondary data analysis*) per mezzo di costosi processi di trasformazione, con lo scopo di individuare le caratteristiche di rilievo dal genoma analizzato. Al termine di questo processo, si ottiene un dato genomico processato che risulta essere di dimensioni più contenute rispetto al dato di tipo *raw*, per quanto riguarda l'occupazione di spazio su disco.
I dati processati sono collezionati da consorzi quali **ENCODE** [9] (Encyclopedia of DNA Element), **TCGA**[1] [28] [10] (The Cancer Genome Atlas), **RoadMap Epigenomics** [24] [18] e **1000 Genomes** [23] [8]; è inoltre consuetudine, per gli autori di articoli di tema biologico, pubblicare i loro dati processati su repository come **GEO** [13] [2](The Gene Expression Omnibus). Questi dati rappresentano una grande ricchezza di informazioni, in quanto sono liberamente accessibili e possono essere utilizzati per ulteriori ricerche. Infine, i dataset sono utilizzati durante l'analisi terziaria dei dati (detta comunemente *tertiary data analysis*) per dare un significato globale a segnali genomici eterogenei, in modo da cercare una risposta anche alle domande biologiche più complesse.

1.1 Situazione attuale

Grandi sforzi di ricerca sono stati dedicati alla produzione di dataset genomici: al contrario, molta meno attenzione è stata concessa alla produzione di una descrizione strutturata dei loro

[1] Il repository TCGA è stato attualmente dismesso e sostiuto con **GDC** (Genome Data Commons), un repository che integra tutti i dati del vecchio portale TCGA con ulteriori repository. Considerando che per questo lavoro sono stati utilizzati i dati provenienti unicamente dal portale dismesso, d'ora in avanti ci riferiremo ad entrambi indistintamente, intendendo riferirsi esclusivamente ai dati TCGA ricavati tramite il portale GDC.

contenuti.
Queste descrizioni, alle quali ci riferiremo con il termine “metadati” e la cui utilità è spesso sottovalutata, risultano invece fondamentali per comprendere come ogni campione biologico sia stato processato, a quali condizioni clinico/biologiche esso sia associato e quali processi tecnologici siano stati impiegati per la sua produzione.
Purtroppo, non esistono standard da rispettare per quanto riguarda la struttura dei metadati: questa mancanza porta, all’interno dei consorzi, alla necessità di applicare regole autonomamente; inoltre, spesso è assente un design concettuale per organizzarli oppure, quando presente, si tratta di uno schema inutilmente complesso e difficile da interpretare.
Riepilogando, ad un crescente interesse per l’analisi terziaria dei dati e a una ricca disponibilità di sorgenti di dati, si contrappone l’assenza di un modello concettuale che permetta di selezionare sorgenti e dataset maggiormente idonei per rispondere a questioni riguardanti il genoma.

1.2 Lavoro proposto

In questa tesi, sono state considerate diverse possibili soluzioni architetturali ed implementative per lo sviluppo di un repository integrato di dati pubblici processati.
Il nostro lavoro si inserisce nel contesto del progetto Data Driven *Genomic Computing* (GeCo), che si concentra sul’analisi terziaria per l’integrazione dei dati genomici. Nel momento in cui questo lavoro di tesi è stato intrapreso, all’interno del progetto era già stato realizzato un prototipo di repository in grado di integrare dati provenienti da tre sorgenti diverse (TCGA, ENCODE e RoadMap Epigenomics) con routine di scaricamento e aggiornamento della struttura: questo prototipo risolve il problema dell’eterogeneità dei dati, rappresentando i metadati in un generico sistema di coppie attributo-valore (tab delimited) e sfruttando soluzioni NoSQL di salvataggio e condivisione dei file. Inoltre, in questo formato, i metadati sono utilizzati per un’iniziale selezione di possibili dataset rilevanti.
Seppur consapevoli del fatto che le coppie attributo-valore possano fornire una soluzione pratica, è necessario ammettere che esse non dispongono di sufficiente semantica per determinate applicazioni.
All’interno di GeCo, dunque, è stato proposto il **Genomic Conceptual Model** [3] (GCM), un modello concettuale per la descrizione di metadati genomici provenienti dalle sorgenti precedentemente elencate. Questa tesi intende proporre una soluzione tecnologica ed architetturale per lo sviluppo e l’implementazione dei concetti presentati in [3], prendendo in considerazione il lavoro precedentemente svolto, gli strumenti di integrazione già implementati e inserendo il framework nel sistema già sviluppato.
In questo lavoro sono state considerate informazioni, provenienti da sorgenti di dati genomici, che descrivono diversi aspetti dello sviluppo di un esperimento: i campioni biologici analizzati, le condizioni di analisi e le tecnologie utilizzate. Queste informazioni, che nelle sorgenti di provenienza possono presentarsi in forma strutturata, semistrutturata o libera, sono generalmente chiamate *metadati*, in quanto non contengono risultati dell’esperimento (chiamati comunemente *dati*), ma ne descrivono semplicemente il contesto. Considerando che questa tesi si concentra solamente sui *metadati*, trascurando gli aspetti dei dati, nel seguito ci si riferirà a questi in

maniera indistinta utilizzando sia il termine "dati" che "metadati".

1.3 Struttura della tesi

I contenuti della tesi sono organizzati in sei capitoli:

Nel Capitolo 2 viene presentato lo Stato dell'arte.

Nel Capitolo 3 viene esposto modello concettuale GCM proposto in [3].

Nel Capitolo 4 sono illustrate le modifiche che hanno permesso di implementare praticamente il modello teorico in una soluzione pronta all'utilizzo.

Nel Capitolo 5 sono dichiarate e motivate le scelte tecnologiche e architetturali attuate per lo sviluppo dell'applicativo.

Nel Capitolo 6 viene proposta una panoramica del funzionamento del framework sviluppato, con esempi pratici realizzati sfruttando una delle sorgenti dati incluse: ENCODE.

Nel Capitolo 7 viene descritta la natura estendibile del framework, illustrando i passaggi necessari per l'eventuale introduzione di nuovi repository, nuovi formati di dati in ingresso e nuovi metodi di trasformazione dei dati.

Infine, nelle conclusioni sono riassunti gli scopi, le valutazioni effettuate e le prospettive future.

Nell'appendice A e B sono descritte, sfruttando il supporto di tabelle illustrative, le regole di mapping per i repository ENCODE e TCGA, rispettivamente.

Nell'appendice C si riporta il modello *xsm* del file di configurazione supportato.

Nell'appendice D le proprietà degli attributi delle tabelle del modello teorico sono messe a confronto con quelle del modello implementato.

Nell'appendice E sono presentate le statistiche usate per valutare le performance del framework analizzate nella Sezione 6.14.

Capitolo 2

Stato dell'arte

2.1 Il sequenziamento del DNA: la Next Generation Sequencing

La genomica, una branca della biologia molecolare il cui scopo è lo studio del genoma[1] degli organismi viventi, è una scienza relativamente recente. Il modello a doppia elica del DNA, attribuito ai premi Nobel James Watson e Francis Crick, è stato pubblicato nell'aprile del 1953 e la prima bozza del genoma umano, scoperto nell'ambito del programma *Human Genome Project*, è stata realizzata nel febbraio del 2001 e completata nell'aprile del 2003. Lo *Human Genome Project*, inizialmente fondato dal National Institutes of Health (NIH), è il risultato di un grande sforzo collettivo compiuto da parte di università e centri di ricerca.
Il sequenziamento del DNA è la determinazione dell'ordine dei diversi nucleotidi (quindi delle quattro basi azotate che li differenziano, cioè Adenina, Citosina, Guanina e Timina) che costituiscono il genoma analizzato. La sequenza di DNA contiene tutte le informazioni genetiche ereditarie che sono alla base dello sviluppo di tutti gli organismi viventi. All'interno di questa sequenza sono codificati i geni peculiari dell'individuo, che interagiscono con fattori di trascrizione (proteine) adibiti alla regolazione dell'espressione genica[2] nel tempo e nello spazio. Determinare la sequenza nucleotidica di un campione risulta quindi utile in ogni campo della biologia e, l'avvento di metodi per il sequenziamento del DNA, ha accelerato significativamente la ricerca. In medicina, ad esempio, il sequenziamento viene impiegato per identificare e diagnosticare malattie ereditarie e per sviluppare nuovi trattamenti terapeutici mirati e minimamente invasivi. In modo simile, lo studio del genoma di agenti patogeni può portare allo sviluppo di metodi di cura contrastanti malattie infettive. Inoltre, la rapidità del processo con cui il processo di sequenziamento può essere effettuato favorisce lo studio del genoma umano su larga scala.
Negli ultimi 15 anni, la tecnologia per il sequenziamento del DNA ha fatto enormi progressi: tra tutti, di rilevanza significativa è l'invenzione della Next Generation Sequencing (NGS o Second Generation Sequencing), una moderna tecnica di diagnostica genetica eseguita sul genoma

[1] Il genoma è la totalità aploide/diploide del DNA contenuto in una cellula di un organismo vivente. Il genoma caratterizza la specie e comprende una parte codificante, ossia i geni, ed una non codificante, le cui funzioni sono poco note.

[2] Con espressione genica si intende il processo attraverso cui l'informazione contenuta in un gene (costituita da DNA) viene convertita in una proteina funzionale.

umano che ha recentemente sostituito la vecchia tecnica adottata, detta *Sanger sequencing*. La NGS viene anche chiamata *high-throughput sequencing* (sequenziamento ad alta resa) perché, a differenza del sequenziamento tradizionale (metodo Sanger), consente di sequenziare, in parallelo, moltissimi frammenti genomici. L'impiego di questa tecnica innovativa permette, in un solo esperimento, di effettuare studi di vario genere, tra i quali la caratterizzazione simultanea di genomi e l'individuazione di riarrangiamenti cromosomici bilanciati e sbilanciati, delezioni e copy number variations (CNV). Nello specifico, le tecniche di NGS permettono di sequenziare:

- **DNA GENOMICO** :
  - Intero genoma
  - Esoma (solo la parte di DNA trascritto in RNA, esoni)
  - Geni mirati
  - Ampliconi (solo prodotti PCR)
- **TRASCRITTOMA**:
  - RNA totale
  - mRNA
  - small RNA (<30 nt)
- **EPIGENOMA**:
  - ChIP-Seq (chromatin immunoprecipitation sequencing: DNA o RNA a cui sono legati specifiche proteine)
  - Metil-Seq (studio del pattern di metilazione del DNA, epigenetica).

Esistono molteplici sistemi NGS, sviluppati da diverse compagnie. Tutti questi sistemi, tuttavia, condividono almeno tre passi fondamentali: la preparazione e immobilizzazione del DNA (cioè la preparazione della cosiddetta sequencing library), la reazione di amplificazione e la reazione di sequenziamento.

2.1.1 Preparazione e immobilizzazione del DNA o Creazione della sequencing Library

Il campione del DNA è ottenuto attraverso un processo di frammentazione casuale del genoma, realizzato inducendo la rottura dei legami fosfodiesterici tra i nucleotidi. Ai frammenti casuali così ricavati, vengono poi aggiunte delle sequenze predefinite, note come 'adattatori' o 'adaptors' e necessarie per ancorare e immobilizzare i frammenti al supporto sul quale avrà luogo la reazione di sequenziamento. I frammenti di DNA, immobilizzati tramite l'aggiunta degli adattatori, costituiscono la cosiddetta libreria di sequenziamento (sequencing library). Esistono almeno tre diversi tipi di adattatori e quindi tre diverse modalità per preparare la sequencing library: adattatori lineari, adattatori circolari e adattatori a bolla. Esistono naturalmente anche tipi di ancoraggio diversi: ad esempio, nel sistema SOLiD i frammenti vengono ancorati ad una lastra in vetro.

2.1.2 Reazione di amplificazione

L'amplificazione può essere indotta in **emulsione** (sistema Roche e sistema SOLiD) o in **soluzione**. Ad esempio, nel sistema GS FLX (Roche) un frammento della sequencing library viene incorporato in una microscopica bolla di acqua assieme a cosiddette **enrichment beads**, delle piccole sfere a cui gli adattatori si possono legare. La reazione di amplificazione (PCR) si compie in questa microbolla acquosa, all'interno della quale il frammento di DNA viene replicato numerose volte. Prodotte le copie clonali del frammento, esse si legano all'enrichment bead ricoprendone la superficie: gli enrichment beads così ottenuti vengono poi depositati su una piastra tecnicamente detta "Picotiter".

2.1.3 Reazione di sequenziamento

Il sequenziamento avviene grazie a complessi meccanismi che, applicati su scala microlitrica, regolano il flusso dei reagenti che vanno a cimentarsi con il DNA immobilizzato. Ogni ciclo di sequenziamento consiste nel cimentare il DNA immobilizzato attraverso una soluzione contenente un nucleotide che, se complementare alla sequenza, viene incorporato: segue un lavaggio e la registrazione dell'evento molecolare appena verificatosi. In particolare, la registrazione dell'evento molecolare avviene attraverso un sistema per immagini.

Come specificato all'inizio, i sistemi NGS sono anche detti high-throughput (ad alta resa) in quanto tutti i frammenti immobilizzati sul supporto di sequenziamento vengono sequenziati in parallelo, a differenza del Sanger sequencing tradizionale in cui si può sequenziare un solo frammento per volta.

Di seguito sono riportati alcuni dettagli riguardanti la modalità di generazione della catena in allungamento, ossia il processo attraverso il quale i nucleotidi vengono incorporati nei diversi sistemi di DNA immobilizzato. Nel pyrosequencing del sistema GS FLX Roche e nel sistema Ion Torrent (Life Technologies) ogni nucleotide viene interrogato singolarmente; al contrario, nei sistemi Illumina, i quattro nucleotidi vengono interrogati in parallelo ma solamente il nucleotide terminatore complementare (terminator nucleotide) si lega, poiché ognuno dei quattro nucleotidi terminatori dispone di un dominio che inibisce il legame degli altri tre; il sistema PacBio RS si basa, invece, sull'utilizzo di nucleotidi che recano un fluoroforo clivabile (rimovibile) che viene staccato durante la reazione di incorporazione; infine, i sistemi SOLiD e CGA utilizzano una modalità di sequenziamento basata sull'utilizzo di oligonucleotidi degenerati, marcati con fluorocromi.

Una volta completato il sequenziamento, i dati vengono analizzati nella fase computerizzata di analisi bioinformatica (alignment, variant calling, filtering and annotation).

2.1.4 Risultati

Le tecnologie di NGS stanno contribuendo positivamente all'incremento di scoperte e conoscenze riguardo a problemi di fondamento biologico e di ricerca clinica, fornendo la possibilità di com-

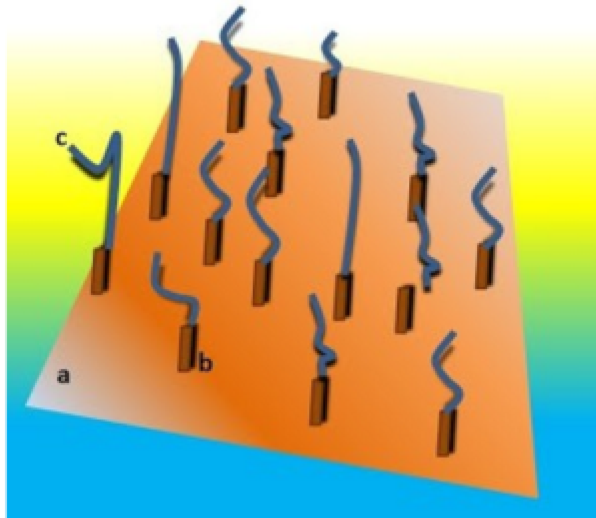


Figura 2.1: Rappresentazione schematica di una sequencing library per la next generation sequencing. a: supporto; b: adattatori; c: frammenti di DNA ancorati al supporto (immobilizzati) tramite gli adattatori.

prendere la caratterizzazione di schemi genomici ed epigenomici e di dare risposta a problemi irrisolti. Tuttavia, esistono questioni ancora aperte, tra cui la modalità di risposta di un gene all'interazione tra DNA e proteine, la struttura della cromatina, la modalità di sviluppo del cancro, piuttosto che il processo attraverso cui caratteristiche genomiche ed epigenomiche influiscono sulla comparsa di sindromi complesse, attribuibili a mutagenesi genomica. Le tecniche di NGS sono volte alla registrazione di caratteristiche quali mutazioni o variazioni del DNA (DNA-seq), transcriptome profiles (RNA-seq), DNA methylations (BS-seq), interazione fra DNA e proteine e stati cromatinici (ChIP-seq and Dnase-seq).
Consorzi di tipo world-wide come ENCODE, TCGA, 100 Genomes Project, Roadmap Epigenomics e tanti altri collezionano la maggior parte di queste caratteristiche prodotte dalla NGS, attraverso dati di tipo *raw*. Una delle più grandi sfide è la cosiddetta *tertiary analysis*, che consiste nel riuscire ad analizzare questa mole di dati [17], al fine di fornirne un fondamento logico.
A causa della molteplicità di laboratori e differenti tecniche di NGS adottate, esistono diversi formati per l'archiviazione dei dati: questo dettaglio implica la presenza di una grande eterogeneità e variabilità di formati di dati tra cui, tra i più diffusi, Bed, NarrowPeak, VCF, SAM, ecc. Da questa constatazione emerge la necessità di integrare e analizzare questi differenti formati di dati [20].

2.2 Genomic Data Model

Molte delle risposte a domande biomediche ancora irrisolte possono essere ritrovate all'interno di collezioni pubbliche di dati eterogenei: tuttavia, i metodi e gli strumenti tutt'oggi disponibili per l'estrazione "del sapere" a partire da questi dati sono ancora troppo miseri e non sufficien-

temente specializzati. Il Genomic Data Model (GDM) è un paradigma di analisi terziaria di differenti tipi di dati (epi)genomici. GDM permette di descrivere, in modo omogeneo, dati semanticamente eterogenei e di creare i presupposti per l'interpretabilità dei dati stessi, raggiungibili attraverso l'utilizzo di un linguaggio dichiarativo ad alto livello, adatto all'interpretazione di *big data* genomici. La combinazione del modello dei dati e del linguaggio di interrogazione fornisce gli strumenti adeguati per l'estrazione delle informazioni dalle sorgenti di dati genomici e permette lo sviluppo di operazioni dominio-specifiche, di tipo data-driven, richieste dalla tertiary data analysis.
GDM è basato sulla nozione di dataset e campioni biologici, descritti mediante due astrazioni fondamentali: una per le regioni genomiche, le quali rappresentano delle porzioni di DNA codificante, con relative caratteristiche, e una per i loro metadati. I dataset sono, quindi, collezioni di campioni, ognuna delle quali è composto da due parti: la regione dei dati, che descrive le caratteristiche dei geni e la loro posizione sul DNA e i metadati, che descrivono proprietà generali del campione. In generale, la principale funzione del GDM è quella di fornire uno schema delle caratteristiche delle regioni genomiche del DNA.

L'aspetto chiave del modello si basa sulla nozione di regione genomica; una regione genomica r_i è una porzione ben definita del DNA, descritta da una quadrupla di valori denominata "coordinate della regione":

$$r_i =< chr, left, right, strand > \tag{2.1}$$

Nella quadrupla, *chr* rappresenta il nome di un determinato cromosoma, *left* e *right* rappresentano le coordinate dirette verso le due estremità del cromosoma ed infine *strand* rappresenta la direzione di lettura di uno dei due filamenti della doppia elica di DNA ed è codificato con +
- o, in caso di assenza, con *.
In accordo con la notazione UCSC, sono state utilizzate la 0-based, half-open inter-base coordinates, quindi, ad esempio, la sequenza genomica considerata è [left, right).
Quindi, una regione r_i corrisponde a tutti i nucleotidi la cui posizione è compresa tra le estremità *left* e *right*; comunque, in generale, non sono incluse sequenze nucleotidiche all'interno della regione di dati, dove piuttosto sono memorizzate proprietà di alto livello della regione, come ad esempio, per campioni biologici peak ChIP-seq, i valori *p* e *q* del peak.
I metadati descrivono le proprietà sperimentali, biologiche e cliniche associate ad ogni campione di dati genomico: data l'alta eterogeneità delle informazioni che possono essere associate ai campioni, nel GDM essi sono rappresentati come coppie arbitrarie di attributo-valore. Da questa premessa ci si aspetta che i metadati descrivano il tipo dell'esperimento o la sequenza dei metodi di analisi (pipeline) sfruttate per la produzione dell'esperimento, le condizioni in cui si è svolto, l'organismo il cui genoma è stato sequenziato e la linea di tessuto o cellule. In caso di studi clinici, inoltre, sono presenti descrizioni individuali che includono i fenotipi.
Formalmente, nel GDM un campione *s* viene descritto attraverso la tripla:

$$s =< id, < r_i, v_i >, m_j > \tag{2.2}$$

Nella tripla, *id* rappresenta l'identificatore del campione (di tipo long); ogni coppia $< r_i, v_i >$ rappresenta una regione, con coordinate r_i e valori v_i che sono attributi tipizzati rappresentanti

id	chr	left	right	strand	*p*-value
1	2	2476	3178	*	0.00000000200
1	2	15 235	15 564	*	0.00000000052
1	5	8790	11 965	*	0.00000000009
1	5	75 980	76 342	*	0.00000000037
2	16	862	923	*	0.00000000018
2	16	1276	1409	*	0.00000000006
2	20	3852	4164	*	0.00000000031

Figura 2.2: Tabella delle regioni.

le caratteristiche della regione. I tipi possono essere BOOLEAN, CHAR, STRING, INT, LONG O DOUBLE e assumono nomi arbitrari. Infine m_j indica coppie attributo-valore, con il secondo elemento (valore) sempre di tipo stringa.
In sintesi, ogni campione *s* ha specifici attributi che descrivono le caratteristiche della sua regione e un set di coppie attributo-valore definite come metadati di *s*. Lo schema della regione dei dati di *s* è l'insieme di tutti gli attributi usati per descrivere le coordinate della regione e i relativi valori, mentre la regione dei tipi dei dati di *s* è descritta come tutti i tipi elementari degli attributi corrispondenti. L'uso di un sistema di tipi per esprimere le regioni rende possibile lo svolgimento di operazioni arbitrarie su valori caratterizzati da tipi compatibili.
Nel GDM, un dataset è una collezione di campioni con lo stesso schema di regione dei dati e dei tipi, il quale contiene valori di regioni conformi con lo schema dei dati proposto e un identificatore che è unico per ciascun dataset; quindi, i dataset sono collezioni omogenee di campioni tipicamente prodotte all'interno dello stesso progetto (presso un centro di ricerca sul genoma o all'interno di un consorzio internazionale) utilizzando la stessa tecnologia e medesimi strumenti.
I dataset sono rappresentabili utilizzando due strutture di dati normalizzate, una per la regione dei dati e l'altra per i metadati. Un esempio contenente un dataset di esperimenti di tipo ChIP-seq è mostrato nelle immagini 2.2 e 2.3. Si noti che l'attributo *id* (prima colonna di ogni tabella) fornisce una connessione molti a molti tra regioni (Immagine 2.2) e metadati (Immagine 2.3) di un campione. Attraverso l'uso di un sistema di dati, tipizzati per esprimere regioni di dati e una coppia arbitraria attributo-valore per i metadati, il GDM fornisce la possibilità di interpretare dataset prodotti usando differenti tecniche sperimentali.

2.3 GenoMetric Query Language

I dati provenienti dalla NGS sono in continuo aumento: da questa costatazione emerge la necessità di rendere utili questi dati e fornire strumenti per analizzarli e ottenerne informazioni.

id	attribute	value
1	antibody_target	CTCF
1	cell	HeLa-S3
1	cell_karyotype	cancer
1	cell_organism	human
1	dataType	ChipSeq
1	view	Peaks
2	antibody_target	JUN
2	cell	H1-hESC
2	cell_organism	human
2	dataType	ChipSeq
2	view	Peaks

Figura 2.3: Tabella dei metadati.

A questo proposito, il GenoMetric Query Language (GMQL), un linguaggio di interrogazioni dichiarativo ad alto livello, è stato creato per rendere accessibili questi dati da parte dei biologi ricercatori e consente di esprimere facilmente query sulle regioni genomiche e sui relativi metadati, in modo simile a quello che può essere ottenuto con l'algebra relazionale e linguaggi SQL. GMQL estende le operazioni algebriche convenzionali con operazioni specifiche del dominio bioinformatico, specificamente progettate per l'applicazione nell'ambito della genomica: esso, quindi, supporta nuove scoperte e un incremento della conoscenza attraverso migliaia o addirittura milioni di campioni, sia per quanto riguarda le regioni che soddisfano le condizioni biologiche sia per il loro rapporto con metadati sperimentali, biologici o clinici. Il nome "GenoMetric Query Language" deriva dalla capacità, da parte del linguaggio, di gestire le distanze genomiche, che sono misurate come basi nucleotidiche tra regioni genomiche (allineate allo stesso riferimento) e calcolate usando operazioni aritmetiche tra le coordinate della regione.

Una query (o programma) GMQL è espressa come una sequenza di operazioni GMQL con la seguente struttura, in cui ogni variabile rappresenta un set di dati GDM:

$$< variable >= operator(< parameters >) < variables > \tag{2.3}$$

Gli operatori si applicano a una o più variabili e costruiscono un risultato che è anch'esso una variabile; i parametri sono specifici per ogni operatore. Parametri di parecchi operatori includono i predicati usati per operazioni di *select* o *join* sui campioni. Tuttavia esistono molteplici tipologie di predicati: tra queste, i predicati complessi, costruiti attraverso espressioni booleane arbitrarie di semplici predicati, come di consuetudine nell'algebra relazionale, i predicati della

regione dati, che devono usare attributi dello schema della regione di dati corrispondente, e, contrariamente a questi, i predicati dei metadati, che possono usare attributi arbitrari. Gli operatori GMQL formano un'algebra chiusa: quindi, i risultati dell'operatore sono espressi come nuovi dataset derivati dai loro operandi e dalle specifiche dell'operatore.
Come è emerso da questo breve excursus, il linguaggio supporta un insieme molto ricco di predicati che descrivono condizioni e proprietà di regioni distali del genoma.

GMQL fornisce un semplice ma potente linguaggio di alto livello che combina procedurali MapReduce con funzionalità ispirate a SQL, richiedendo minime conoscenze nell'ambito informatico. GMQL cambia il paradigma riguardante la modalità di gestione di dati NGS, fornendo operazioni unarie standard di SELECT, ORDER, AGGREGATE, PROJECT e MERGE e operazioni binarie di UNION e DIFFERENCE; inoltre, fornisce operazioni dominio-specifiche tra cui COVER, JOIN e MAP: queste operazioni hanno un'interpretazione biologica.
Una tipica query GMQL inizia con un'operazione di SELECT, la quale crea un dataset con i soli campioni di dati filtrati da un dataset in input utilizzando un predicato sugli attributi dei metadati. Successivamente, la query processa i campioni, selezionati attraverso operazioni effettuate sulla loro regione di dati e/o sui metadati. Infine, attraverso un'operazione di MATERIALIZE, viene memorizzato il dataset salvando la regione dei dati, per ognuno dei suoi campioni, in un singolo file standard GTF e i relativi metadati in un file testo di tipo *tab delimited*.
GMQL consente ai biologi di interrogare facilmente i dati della regione genomica di NGS e i loro metadati, poiché si tratta del primo linguaggio in grado di includere i metadati nel processo di calcolo e, inoltre, in grado di supportare la gestione dei metadati. In questo modo, i metadati sono coinvolti sia nella selezione che nell'abbinamento e vengono anche "trasportati" lungo il processo, così da essere accessibili anche dopo l'esecuzione della query. Ad oggi, GMQL include dati sperimentali da ENCODE e TCGA2BED, fornendoli in un formato GDM.

2.4 Sorgenti di dati: Encode e TCGA (GDC)

Il Cancer Genome Atlas (TCGA) [30] è uno dei maggiori depositi di dati pubblici di genomica, epigenomica e proteomica per più di 30 diverse tipologie di cancro (http://www.cancergenome.nih.gov/). TCGA include vari tipi di dati sperimentali Next Generation Sequencing (NGS) [15] [31] [26], come Copy Number Variation (CNV) [7], DNA-methylation [5] bird2002dna, DNA-sequencing (DNA-seq) [4] includendo mutazioni del genoma e dell'esoma, Gene expression (RNA-seq V1, RNA-seq V2) [21] [19], microRNA sequencing (miRNA-seq) [32] e i loro metadati (informazioni cliniche o di tipo *biospecimen*). I dati forniti da TCGA hanno arricchito significativamente molti studi sul cancro: infatti, creando una pipeline di analisi genomica dei dati, TCGA analizza i tessuti umani su una scala molto ampia, al fine di raccogliere e selezionare il maggior numero possibile di mutazioni genomiche. Fino al giugno 2016 i dati e metadati sperimentali di TCGA erano consultabili liberamente e legalmente sul portale dei dati TCGA mentre, in tempi recenti, la totalità dei dati è stata trasferita sul portale "The Genomic Data Commons (GDC)", una piattaforma di condivisione dei dati che promuove la medicina di precisione in oncologia (https: //gdc.nci.nih.gov/) e sulla quale sono disponibili i dati TCGA originali considerati in questo lavoro di tesi. TCGA fornisce un'ampia collezione di dati relativi a malattie

tumorali e controlli raccolti da miglialia di pazienti, che contribuiscono in maniera determinante alla generazione del più grande repository contenente questa tipologia di dati: questa sorgente mette a disposizione una vasta gamma di strumenti per l'analisi e l'estrazione di nuove informazioni riguardanti diverse tipologie tumorali. Tuttavia, al fine di sfruttare appieno questo grande archivio di dati, sono necessari nuovi metodi per la standardizzazione del formato, la gestione, l'integrazione e l'interrogazione dei dati forniti, con lo scopo di favorire innovazioni utili al trattamento del cancro.

Encyclopedia of DNA Elements (ENCODE) [9] è un consorzio fondato dalla National Human Genome Research Institute (NHGRI) che ha accettato diverse collaborazioni con molteplici gruppi di ricerca internazionali. Il suo obiettivo è quello di generare una lista comprensibile di elementi funzionali del genoma umano. ENCODE esegue un gran numero di studi sfruttando tecniche NGS per mappare gli elementi funzionali attraverso il genoma umano e fornisce tutti i dati e le descrizioni dei protocolli, resi pubblicamente disponibili attraverso l'apposito sito web [11].

2.5 Extract, transform, load (ETL)

Extract, transform, load (ETL) è una tecnica per l'integrazione di dati provenienti da più origini ed è composta da tre passaggi fondamentali: l'estrazione dei dati, la trasformazione dei dati estratti e il caricamento dei dati trasformati. Il processo ETL è diventato un concetto popolare negli anni '70. L'estrazione dei dati comprende il processo attraverso cui i dati vengono estratti da fonti dati omogenee o eterogenee; la trasformazione dei dati è l'operazione attraverso cui i dati vengono trasformati al fine di consentirne l'archiviazione nel formato o nella struttura appropriata per una conseguente interrogazione o analisi degli stessi; il caricamento è il processo in cui i dati vengono caricati nel database di destinazione finale, solitamente rappresentato da un archivio dati operativo, *data mart* o *data warehouse*. Poiché l'estrazione dei dati richiede parecchio tempo, è buona norma eseguire le tre fasi in parallelo: infatti, mentre i dati vengono estratti, un altro processo di trasformazione viene eseguito per l'elaborazione dei dati già ricevuti e per la loro preparazione al caricamento. Il caricamento dei dati può iniziare senza attendere il completamento delle fasi precedenti. Generalmente i sistemi ETL integrano dati provenienti da più applicazioni (sistemi), sviluppati e supportati da diversi fornitori o ospitati su macchine fisicamente differenti.

2.5.1 Extract

La prima fase di un processo ETL comporta l'estrazione dei dati dai sistemi di origine. In molti casi, questo processo risulta essere l'aspetto più importante dell'ETL, dal momento che l'estrazione dei dati decreta le condizioni per l'eventuale successo o fallimento dei processi successivi. La maggior parte dei progetti di *data-warehousing* combina dati provenienti da diversi sistemi di origine; in aggiunta, ogni sistema separato può utilizzare una diversa organizzazione e/o formato dei dati. I formati di origine dati comuni includono database relazionali, XML, JSON e file flat, ma possono anche includere strutture di database non relazionali quali IMS

(Information Management System), altre strutture di dati come Virtual Storage Access Method (VSAM) o Indexed Sequential Access Metodo (ISAM), oppure formati recuperati da fonti esterne con mezzi come lo spidering[3] o screen-scraping[4]. Uno *stream* diretto dei dati estratti dalla sorgente, con conseguente caricamento *on-the-fly* dei dati sulla destinazione è un metodo alternativo di sviluppare la tecnica ETL quando non è richiesta alcuna memorizzazione di dati intermedi. In generale, la fase di estrazione mira a convertire i dati in un unico formato che risulti appropriato all'elaborazione del conseguente processo di trasformazione.

Una fase intrinseca dell'estrazione implica la convalida dei dati, per verificare che i dati estratti dalle origini abbiano i valori corretti/previsti in un dato dominio. Nell'eventualità in cui i dati non rispettino le regole di convalida, essi vengono interamente oppure parzialmente respinti. I dati rifiutati possono anche essere segnalati al sistema di origine, al fine di effettuare analisi più approfondite sulle cause dell'errore e di identificare e correggere eventuali errori presenti nel software di generazione degli stessi. In alcuni casi, lo stesso processo di estrazione potrebbe essere indotto a verificare le regole di convalida dei dati per validarli prima di permetterne il passaggio alla fase successiva.

2.5.2 Transform

Durante la fase di trasformazione, una serie di regole o funzioni applicate ai dati estratti svolgono il processo di preparazione degli stessi per il caricamento sul dispositivo target. Esistono alcuni dati che non richiedono alcuna trasformazione: essi sono noti come "direct move" o "pass through" data. Un'importante funzione di trasformazione è la pulizia dei dati che mira a trasmettere solamente i dati considerati "corretti" (l'interpretazione del termine varia a seconda delle specifiche date).
Su questo fronte, la maggior sfida consiste nella definizione di un'interfaccia comune per far interagire sistemi diversi.

2.5.3 Load

Nella fase di caricamento, i dati trasformati sono trasferiti verso un target finale, che può variare da un semplice file flat ad un *data warehouse*. Questo processo è soggetto ad ampie variazioni a seconda delle esigenze dell'organizzazione: alcuni *data warehouse* possono sovrascrivere le informazioni esistenti con informazioni cumulative attraverso un processo di aggiornamento dei dati estratti, spesso effettuato su base giornaliera, settimanale o mensile; altri *data warehouse* (o anche altre parti dello stesso *data warehouse*) possono aggiungere nuovi dati in una forma storica a intervalli regolari.
Per esempio, si consideri un *data warehouse* in cui è necessario mantenere i record delle vendite dell'ultimo anno: questo sistema sovrascrive tutti i dati più vecchi di un anno con dati più recenti;

[3]Lo spider (o crawler) è un software che analizza i contenuti di una rete (o di un database) in un modo metodico e automatizzato, in genere per conto di un motore di ricerca. Uno spider è un tipo di bot (programma o script che automatizza delle operazioni), e gli spider solitamente acquisiscono una copia testuale di tutti i documenti visitati e le inseriscono in un indice.

[4]Con screen-scraping si intende la collezioni di dati visualizzati a schermo da una sorgente (per esempio una pagina web) ottenuti parsando il contenuto della pagina

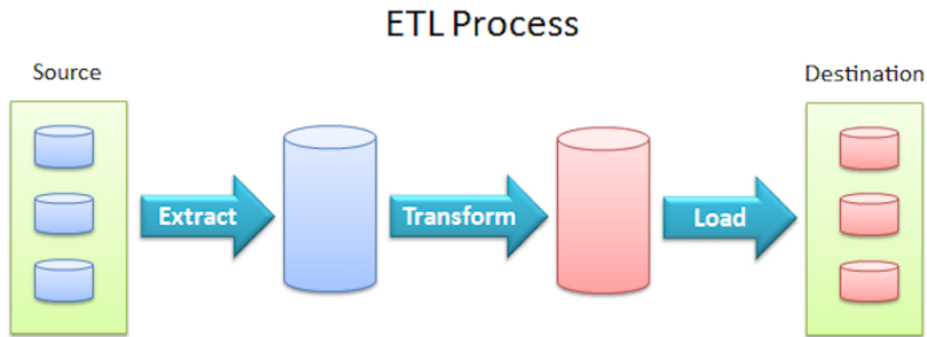


Figura 2.4: Architettura di un sistema ETL

tuttavia, l'inserimento dei dati per una finestra temporale di un anno viene effettuato in modo storico. I tempi e gli obiettivi da sostituire o aggiungere sono scelte progettuali strategiche che dipendono dal tempo disponibile e dalle esigenze inoltre, sistemi più complessi possono conservare una cronologia e una traccia per il controllo di tutte le modifiche ai dati caricati nel *data warehouse*.
Poiché la fase di caricamento interagisce con il database, in questo processo vengono applicati i vincoli definiti nello schema della base di dati (per esempio unicità, integrità referenziale, campi obbligatori), che contribuiscono anche alla prestazione di qualità complessiva dei dati del processo ETL.

2.6 Sistemi di integrazione di dati biologici già esistenti

Molte ricerche hanno affrontato il problema dell'accesso a fonti di dati genomici eterogenei. Un primo lavoro di analisi per le sorgenti di dati genomici è presentato in [16] che, oltre a fornire uno studio dei sistemi al tempo sviluppati per l'integrazione dei dati genomici elencandone i pregi e i difetti, delinea le tre principali metodologie impiegate per l'implementazione di un sistema di integrazione dati: **warehouse integration**, **mediator-based integration** e **navigational integration**.

La tecnica della **warehouse integration** consiste nella creazione di una base di dati propria dove, per mezzo di un processo di integrazione (per esempio ETL), i dati vengono scaricati, elaborati ed infine caricati sul repository condiviso.

Il **mediator-based integration** risolve il problema dell'integrazione dati fornendo una struttura di traduzione delle query da un linguaggio comune al linguaggio della sorgente. I dati non sono salvati in un sistema di integrazione: questo evita lo scaricamento e il caricamento periodico dei dati poiché consente di mantenerli sempre aggiornati all'ultima versione senza alcuno sforzo, sfruttando un sistema di recupero dati, conosciuto come *on the fly*. I sistemi mediator-based integration si suddividono a loro volta in *global-as-view* (GAV) e *local-as-view* (LAV): i

secondi, anche se più difficili da implementare, forniscono una soluzione più scalabile in caso di aggiunta di una nuova sorgente di dati.

Infine, l'approccio **navigational integration** (o link-based integration) nasce dalla crescente difficoltà di navigazione delle pagine web per ottenere le informazioni desiderate, a causa di un numero di sorgenti sempre crescente: essenzialmente, i percorsi specifici costituiscono un workflow in cui uno o più output di una fonte o uno strumento sono reindirizzati come input alla sorgente successiva, finché l'informazione non è stata ottenuta. In senso pratico, le query sono trasformate in espressioni, possibilmente diverse, che specificano il percorso i cui risultati possono soddisfare le query a diversi livelli.

Fra tutte queste soluzioni, quella che sembra rispondere meglio alle esigenze di un sistema di integrazione è l'approccio data **warehouse integration** [14] [16]: infatti, questo approccio permette di ottimizzare il processo di gestione e interrogazione dei dati in quanto, per il suo utilizzo, viene definito uno schema comune delle sorgenti; inoltre, è possibile filtrare le informazioni in ingresso, aggiungendo la possibilità di annotare i dati a piacimento e realizzando anche un processo di validazione dei dati. In aggiunta, considerando che i dati sono salvati localmente, vengono eliminati tutti i problemi di congestione della rete, possibili colli di bottiglia e non disponibilità del servizio nei quali si potrebbe incorrere utilizzano direttamente servizi offerti da terze parti. D'altra parte, cambiamenti della sorgente potrebbero determinare necessità di modifica del processo di importazione: per evitare questo inconveniente, è opportuno quindi realizzare un processo che sia, per quanto possibile, indipendente dalla sorgente. Per giunta, dovendo attuare un processo per l'integrazione dei dati, ci si potrebbe imbattere nel problema di dati non aggiornati: per questa ragione, il processo di integrazione dei dati deve essere eseguito saltuariamente. Un ulteriore vantaggio potrebbe risiedere nella possibilità di tenere uno storico dei dati delle diverse sorgenti, funzionalità non implementabile con gli altri due approcci.
Esempi applicativi di utilizzo di modelli ER (e diagrammi UML) per la rappresentazione di entità genomiche o bioinformatiche vengono illustrati in [6].
Molti lavori hanno precedentemente proposto un modello concettuale per la rappresentazione di entità biologiche e per la loro interazione: fra tutti quelli analizzati, il nostro approccio è simile a quello avanzato da [1], in quanto proponiamo un modello concettuale per supportare il continuo processo di integrazione dei metadati e per offrire un'interfaccia ad alto livello che sia comune per la loro interrogazione e la rilevazione di dataset rilevanti. Come per [1], nascondiamo le sorgenti dei dati fornendo un'interfaccia facile da utilizzare ma, differentemente, nel nostro lavoro si considerano anche esperimenti di tipo molto diverso: mentre in [1] sono considerati unicamente sorgenti di dati epigenomici (come ENCODE e Roadmap), nel modello proposto sono presi in esame anche dati offerti da portali come TCGA, in cui sono presenti esperimenti più incentrati sull'espressione genica, mutazioni e variazioni. Inoltre, lo schema e l'infrastruttura indicati sono sufficientemente generici per accogliere in futuro dati di altre tipologie che attualmente non sono stati ancora presi in considerazione.

Capitolo 3

Integrazione dei dati genomici

3.1 Genomic Conceptual Model

Con l'espressione *Genomic Conceptual Model*(GCM) [3] si intende un modello concettuale utilizzato per la descrizione di metadati genomici delle sorgenti di dati. Il GCM è incentrato sulla nozione di *experiment item*, tipicamente rappresentato da un file contenente regioni genomiche e le loro proprietà, che viene analizzato da tre punti di vista:

- La tecnologia utilizzata nell'esperimento, incluse informazioni sul contenitore dell'*item* e sul suo formato.
- Il processo biologico osservato nell'esperimento, in particolare il campione sequenziato e la sua provenienza, indicata dal donatore.
- La gestione dell'esperimento (organizzazioni o progetti che sono responsabili del suo svolgimento).

Lo schema concettuale è stato costruito applicando un approccio *top-down* basato sull'analisi sistematica degli attributi dei metadati e delle loro proprietà in diverse sorgenti di dati genomici. In seguito, esso è stato verificato *bottom-up* utilizzando TCGA, ENCODE e GEO: questo processo ha dimostrato che lo schema ER, che descrive queste sorgenti, può essere costruito come sottoinsieme del GCM.

3.2 Design

Molte sorgenti di dati forniscono un'interfaccia per l'estrazione dei metadati: essa è costituita da un semplice template per effettuare query oppure da un'interfaccia di programmazione di applicazioni (API) che permettono la selezione di dati sperimentali. Alcune sorgenti inoltre forniscono una descrizione tabulare dei metadati, in modo da permettere interrogazioni più sistematiche o in formati semistrutturati come file XML o JSON.

Tabella 3.1: *Tassonomia degli attributi e valori ortogonali*

Livello	Simbolo	Proprietà	Default
Sorgente	C	Contestuale	Non Contestuale
	D	Dipendente	Indipendente
	R	Ristretto	Libero
	S	Valore Singolo	Multivalore
	M	Obbligatorio	Opzionale
Repository	H	Manuale	Estratto
Integrato	O	Ontologico	Ordinario

3.2.1 Tassonomia degli attributi

Di seguito è presentata la tassonomia delle principali proprietà degli attributi dei metadati: essi saranno poi applicati ad ogni sorgente considerata, così da caratterizzare in maniera completa il suo contenuto. Gli attributi possono essere:

- **Contestuali** (C), quando sono presenti solo in specifici contesti, tipicamente quando un altro attributo assume uno specifico valore. In questi casi esiste una *dipendenza di esistenza* tra i due attributi.

- **Dipendenti** (D), quando il dominio dei loro possibili valori è ristretto, tipicamente quando un altro attributo assume uno specifico valore. In questi casi esiste una *dipendenza di valore* tra i due attributi.

- **Ristretti** (R), quando il loro valore deve essere scelto da un vocabolario controllato.

- **Singolo Valore** (S), quando essi assumono al più un valore per ogni specifico esperimento.

- **Obbligatori** (M), quando devono assumere obbligatoriamente un valore diverso dal valore nullo, sia per ogni esperimento sia all'interno di uno specifico contesto[1].

La tassonomia risultante è mostrata nella tabella 3.1, che include anche le proprietà ortogonali rispetto a quelle precedentemente elencate.
Di default, gli attributi non hanno alcuna delle proprietà precedentemente elencate, ma si considerano verificate le caratteristiche ad essi ortogonali.
Pochi attributi risultano essere obbligatori e, sfortunatamente, le diverse sorgenti da cui provengono non concordano su quali lo siano: in molti casi sono indicati ed etichettati come obbligatori attributi diversi in ciascuna delle fonti considerate.
Queste cinque categorie sono state impiegate per descrivere gli attributi che sono inclusi nel modello concettuale, come dichiarato nella seguente sezione: inoltre, tutti gli attributi sono stati

[1] Con "specifico contesto" si intende che la presenza dell'obbligatorietà può essere definita anche solo in caso di particolari valori assunti da altri attributi (quindi in combinazione con regole di contestualità o dipendenza), mentre negli altri casi questa proprietà potrebbe non valere

etichettati con un vettore di proprietà. Ad esempio,

$$Type^{[RSM]} \tag{3.1}$$

denota *Type* come un attributo ristretto, a valore singolo e obbligatorio, mentre

$$Pipeline^{[D(Technique)S]} \tag{3.2}$$

denota che l'attributo *Pipeline* è a valore singolo e dipendente da un altro attributo, in questo caso *Technique*.

3.2.2 Source Analysis

Per lo svolgimento di questo lavoro sono state analizzate diverse sorgenti: fra tutte, sono state selezionate TCGA e ENCODE poiché forniscono le collezioni più comprensibili di attributi dei metadati.

- **TCGA** fornisce molti attributi di metadati specifici per tipologia di pipeline; da questo repository sono stati selezionati 22 attributi, comuni a tutte le pipeline, che sono risultati i più interessanti da un punto di vista biologico.
- **ENCODE** prevede la possibilità di ottenere una lista espansa di metadati, oppure una sua forma più breve: nella forma espansa sono presenti più di 2000 attributi, mentre nella lista breve sono contenuti 49 attributi per gli esperimenti, 44 attributi per i campioni biologici e 28 attributi per le descrizioni dei file.

3.2.3 Altre proprietà

Di seguito sono definite altre proprietà che non possono essere osservate nelle sorgenti ma che caratterizzano gli attributi dei metadati del nostro repository integrato:

- **Manuale** (H) quando il valore è fornito dal gestore del repository.
- **Ontologico** (O) quando è prevista, come sviluppo futuro, un'interfaccia che supporta ricerche basate su similarità sintattiche e semantiche[2].

3.2.4 Regole

Le regole sono funzionali all'espressione dell'esistenza e della dipendenza tra i vari valori.

- **Dipendenza di esistenza**: *Technique* = "Chip-seq" $\rightarrow$ M(*Target*) indica che *Target* è obbligatorio se *Technique* assume come valore "Chip-seq", mentre *Technique* $\neq$ "Chip-seq" $\rightarrow$ NULL(*Target*) indica che se *Technique* è diverso da "Chip-seq" il valore di *Target* non è specificato, infine $\exists$ *Technique* $\rightarrow$ NULL(*Target*) indica che se *Technique* è diverso dal valore nullo allora *Target* non assume alcun valore.

[2]Questa proprietà non è di diretto interesse per questo lavoro di tesi, ma verrà sarà mostrata una struttura predisposta utile ad una futura estensione del progetto.

- **Dipendenza di valore**: *Datatype* = "raw data" $\rightarrow$ *Format* = "fastq" indica che se *Datatype* assume valore "raw data" allora *Format* dovrà essere "fastq", mentre *Datatype* = "raw data" $\rightarrow$ *Format* $\neq$ "fastq" indica che se *Datatype* assume valore "raw data" allora *Format* non può assumere valore "fastq".

3.3 Modello ER

Di seguito è presentato il modello entity-relationship, che è stato introdotto in [3] e utilizzato come modello teorico ideale per la rappresentazione dei dati genomici.
Sarà mostrato come sia stato necessario un adattamento del modello teorico alla realtà pratica dei diversi repository analizzati, e le modifiche applicate al modello saranno elencate utilizzando un grafico di confronto e giustificate attraverso l'illustrazione di esempi pratici acquisiti dai repository.

3.3.1 Modello Ideale

Il modello è basato sul concetto centrale di **Item**[3] che rappresenta l'unità elementare dell'esperimento. Dall'entità centrale sono stati sviluppati tre sottoschemi che richiamano lo schema a stella tipico delle *data warehouse*. I sottoschemi descrivono rispettivamente gli aspetti biologici, tecnologici e di gestione dell'esperimento esaminato nell'item.

Unità centrale

A questo punto, risulta opportuno descrivere gli attributi dell'entità **Item** e il vettore delle proprietà per ognuno di essi.
Gli attributi $SourceId^{[SM]}$ e $Datatype^{[RSM]}$ denotano rispettivamente l'identificatore dell'item della sorgente analizzata e il tipo di dato dell'item; queste informazioni sono obbligatorie, quindi i due attributi devono essere sempre presenti e avere un singolo valore.
$Format^{[D(Datatype)RSM]}$ specifica il formato dell'item (es. ["fastq", "wiggle", "bed", "tsv", "vcf", "maf", "xml"]) e dipende da *Datatype*. Ad esempio, un formato "bed" è compatibile con una tipologia di dato "peak", ma non compatibile con "read".
Altri attributi sono $Size^{[SM]}$, $SourceUrl^{[M]}$, $LocalUrl^{[C(Format)SM]}$ e $Pipeline^{[D(Technique)S]}$.

$$\mathbf{Item}.DataType = \textit{"raw data"} \rightarrow \mathbf{Item}.Format = \textit{"fastq"} \tag{3.3}$$

L'attributo *Pipeline*, dipendente dalla tecnica, descrive i processi utilizzati durante l'esperimento per la processazione dei dati, determinando il tipo e il formato dell'item prodotto in output.
Nello schema è definito l'attributo multivalore *SourceUrl* che ha il compito di tenere traccia della provenienza dello stesso item da repository diversi.
Infine, l'attributo *LocalUrl* indica la locazione fisica del file salvato all'interno del repository

[3] In grassetto si intendono le entità, in corsivo gli attributi e senza alcuna enfasi le singole istanze

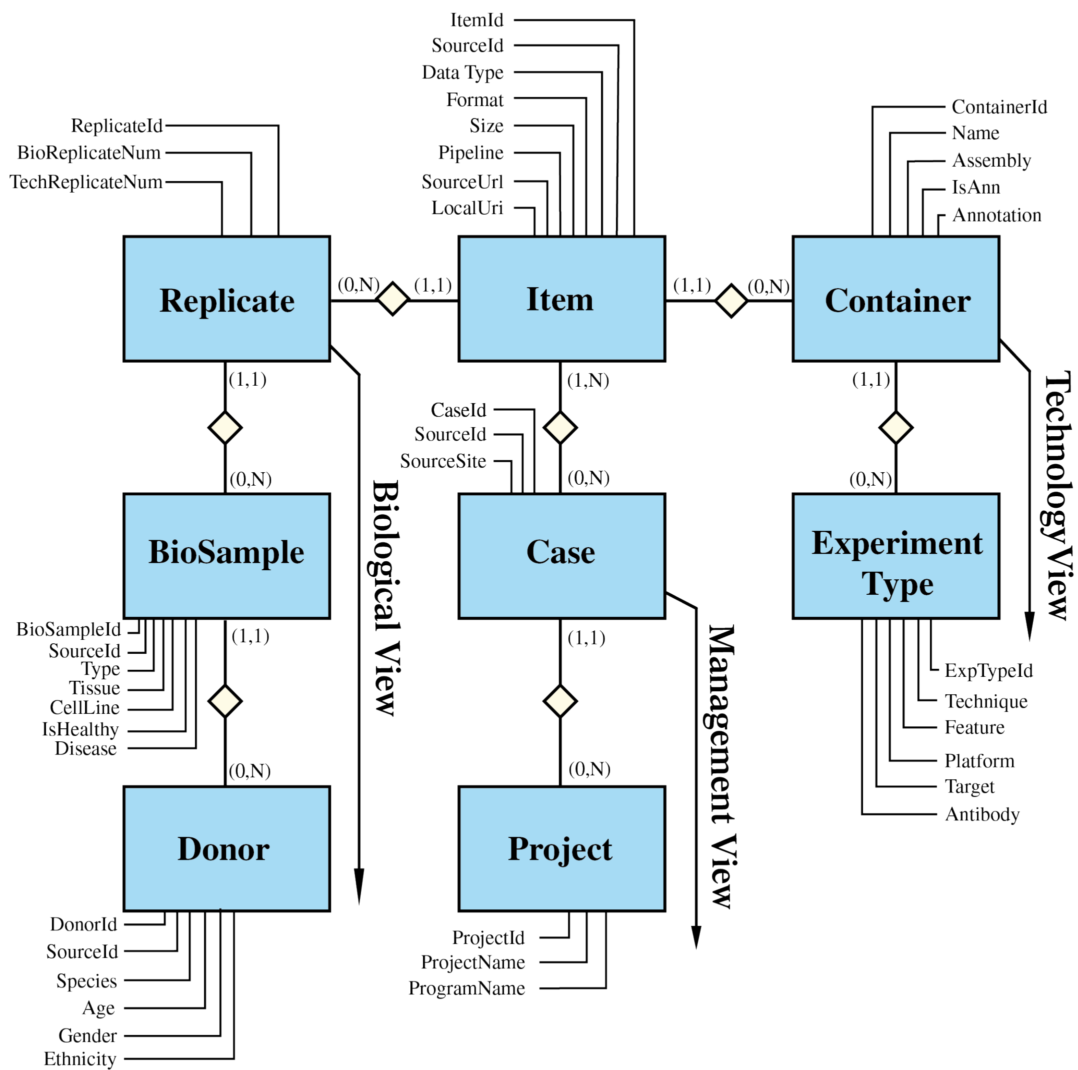


Figura 3.1: Diagramma ER del GCM

integrato. Se il formato del file è di tipo processato (*bed*) allora questo deve essere inserito nella base di dati, e deve quindi esistere un url locale di riferimento al dato:

$$\textbf{Item}.Format = \textit{"bed"} \rightarrow M(\textbf{Item}.LocalUrl) \tag{3.4}$$

Vista biologica

Questo sottoschema consta di una catena di entità **Item** - **Replicate** - **Biosample** - **Donor**. La catena di informazioni riporta il processo biologico atto alla produzione dell'item. Tutte le relazioni sono del tipo "uno a molti", in quanto ogni Item è associato ad un dato replicato (Replicate) tecnico, che è a sua volta associato ad un unico campione (BioSample) di tessuto o di coltura cellulare che deriva, infine, da un unico donatore (Donor).
Il **Donor** rappresenta lo specifico organismo da cui deriva il materiale biologico: esso ha un $SourceId^{[SM]}$ che ne contraddistingue l'unicità all'interno del repository ed è inoltre caratterizzato da una specie ($Species^{[RSM]}$), un'età ($Age^{[S]}$), un sesso ($Gender^{[RS]}$) e un'etnia ($Ethnicity^{[RS]}$). Possibili regole di dipendenza di valore sono

$$Donor.Species = \textit{"Homo sapiens"} \rightarrow Container.Assembly \in \textit{["GRCh38","hg19","hs37d5"]} \tag{3.5}$$

$$Donor.Species = \textit{"Mus musculus"} \rightarrow Container.Assembly \in \textit{["mm9","mm10","GRCm38"]} \tag{3.6}$$

L'entità **BioSample** descrive il campione biologico ottenuto dal donatore e utilizzato per l'esperimento. Ha anch'esso un $SourceId^{[M]}$ funzionale all'identificazione univoca del campione all'interno della sorgente. La ragione per la predisposizione di molteplici valori per *SourceId* risiede nell'eventualità di dover identificare univocamente, con un solo elemento all'interno del repository unificato, i campioni biologici dei diversi repository che rappresentano l'informazione di un unico campione esistente nel mondo reale.
L'attributo $Type^{[RSM]}$ è ristretto ai valori ["cell-line", "tissue"]. In base al valore di *Type* uno solo fra gli attributi $Tissue^{[C(Type)SMO]}$ e $CellLine^{[C(Type)SMO]}$ diventa obbligatorio: la condizione è espressa dal seguente sistema di regole:

$$\textbf{BioSample}.Type = \textit{"tissue"} \rightarrow M(\textbf{BioSample}.Tissue) \tag{3.7}$$

$$\textbf{BioSample}.Type = \textit{"cell-line"} \rightarrow M(\textbf{BioSample}.CellLine) \tag{3.8}$$

$$\exists \textbf{BioSample}.CellLine \rightarrow Null(\textbf{BioSample}.Tissue) \tag{3.9}$$

$$\exists \textbf{BioSample}.Tissue \rightarrow Null(\textbf{BioSample}.CellLine) \tag{3.10}$$

La propietà booleana $IsHealty^{[RS]}$ esprime lo stato di salute del donatore al momento dell'estrazione del campione.
L'attributo $Disease^{[C(IsHealty)D(Tissue)O]}$ dipende contestualmente da *IsHealthy*

$$\textbf{BioSample}.IsHealthy \rightarrow Null(\textbf{BioSample}.Disease) \tag{3.11}$$

e può essere multivalore; inoltre esso è dipendente da *Tissue*

$$\textbf{BioSample}.Tissue = \text{"liver"} \rightarrow \textbf{BioSample}.Disease \in \text{["viral hepatitis","liver lymphoma",...]} \quad (3.12)$$

$$\textbf{BioSample}.Tissue = \text{"liver"} \rightarrow \textbf{BioSample}.Disease \notin \text{["acute leukemia","pilorus cancer",...]} \quad (3.13)$$

in quanto una data malattia può essere tessuto-specifica e quindi interessare esclusivamente un determinato tessuto istologico.
Gli attributi *Tissue, CellLine* e *Disease* sono stati contrassegnati come Ontologici con l'intenzione, in previsioni di sviluppi futuri, di esprimere il valore di questi elementi anche attraverso loro sinonimi o generalizzazioni in campo medico, in modo da facilitarne la ricerca all'interno del repository [12].
L'entità **Replicate** rappresenta l'operazione di suddivisione dello stesso campione biologico in sotto-campioni. Questa entità è particolarmente rilevante in sorgenti di dati epigenomici, quali ENCODE, che differenziano fra replicati biologici e tecnici, mentre questa distinzione non è presente nella maggior parte delle altre sorgenti osservate.

Vista tecnologica

In questa vista la catena di entità **Item** - **Container** - **ExperimentType** descrive le tecnologie atte allo svolgimento dell'esperimento.
L'entità **Container** descrive proprietà comuni di item omogenei, ossia condivisori della stessa struttura dati e generati dallo stesso tipo di esperimento.
Gli attributi includono $Name^{[SM]}$ e $Assembly^{[C(Datatype)D(Species)RSM]}$. *Assembly* assume valore solo per item di un particolare *Type*

$$\textbf{Item}.DataType \in \text{["aligned read","peak","signal"]} \rightarrow M(\textbf{Container}.Assembly) \quad (3.14)$$

ed è ristretto ad un vocabolario limitato in accordo con l'attributo *Species*

$$\textbf{Donor}.Species = \text{"Homo sapiens"} \rightarrow \textbf{Container}.Assembly \in \text{["GRCh38","hg19","hs37d5"]} \quad (3.15)$$

$$\textbf{Donor}.Species = \text{"Mus musculus"} \rightarrow \textbf{Container}.Assembly \in \text{["mm9","mm10","GRCm38"]} \quad (3.16)$$

L'attributo booleano $IsAnn^{[RSM]}$ è utilizzato per distinguere items sperimentali attraverso annotazioni conosciute: quando assume valore positivo, $Annotation^{[C(IsAnn)RSM]}$ deve esistere

$$\textbf{Container}.IsAnn = \text{"true"} \rightarrow M(\textbf{Container}.Annotation) \quad (3.17)$$

mentre quando assume valore negativo, *Annotation* deve assumere valore nullo

$$\textbf{Container}.IsAnn = \text{"false"} \rightarrow Null(\textbf{Container}.Annotation) \quad (3.18)$$

Annotation dispone di un vocabolario limitato che include i termini "Gene", Exon", "TSS", "Promoter", "Enhancer", "Cpg-Island".
ExperimentType denota l'entità che descrive gli specifici metodi utilizzati per la produzione di ogni item: esso prevede l'attributo obbligatorio $Technique^{[RSM]}$ e l'attributo obbligatorio e curato manualmente $Feature^{[D(Technique)RSMH]}$, al fine di descrivere le specifiche caratteristiche dell'esperimento.
Il valore $Platform^{[C(Datatype)RSM]}$ illustra la tecnologia di *Next Generation Sequencing* utilizzata per il processo di sequenziazione e dipende dal *Datatype* dell'item

$$\mathbf{Item}.DataType = \textit{"raw data"} \rightarrow M(\mathbf{ExperimentType}.Platform) \tag{3.19}$$

Quando la ***Technique*** assume valore "Chip-seq", i due attributi $Target^{[C(Technique)RSM]}$ e $Antibody^{[C(Technique)D(Target)RSM]}$ non possono assumere valore nullo

$$\mathbf{ExperimentType}.Technique = \textit{"Chip-seq"} \rightarrow M(\mathbf{ExperimentType}.Target) \tag{3.20}$$

$$\mathbf{ExperimentType}.Technique = \textit{"Chip-seq"} \rightarrow M(\mathbf{ExperimentType}.Antibody) \tag{3.21}$$

mentre quando il valore di *Technique* non assume questo valore, sia *Target* che *Antibody* devono assumere un valore nullo

$$\mathbf{ExperimentType}.Technique \neq \textit{"Chip-seq"} \rightarrow Null(\mathbf{ExperimentType}.Target) \tag{3.22}$$

$$\mathbf{ExperimentType}.Technique \neq \textit{"Chip-seq"} \rightarrow Null(\mathbf{ExperimentType}.Antibody) \tag{3.23}$$

L'attributo *Antibody* dipende dal valore di *Target*, in quanto esso è specifico per ogni determinato antigene.

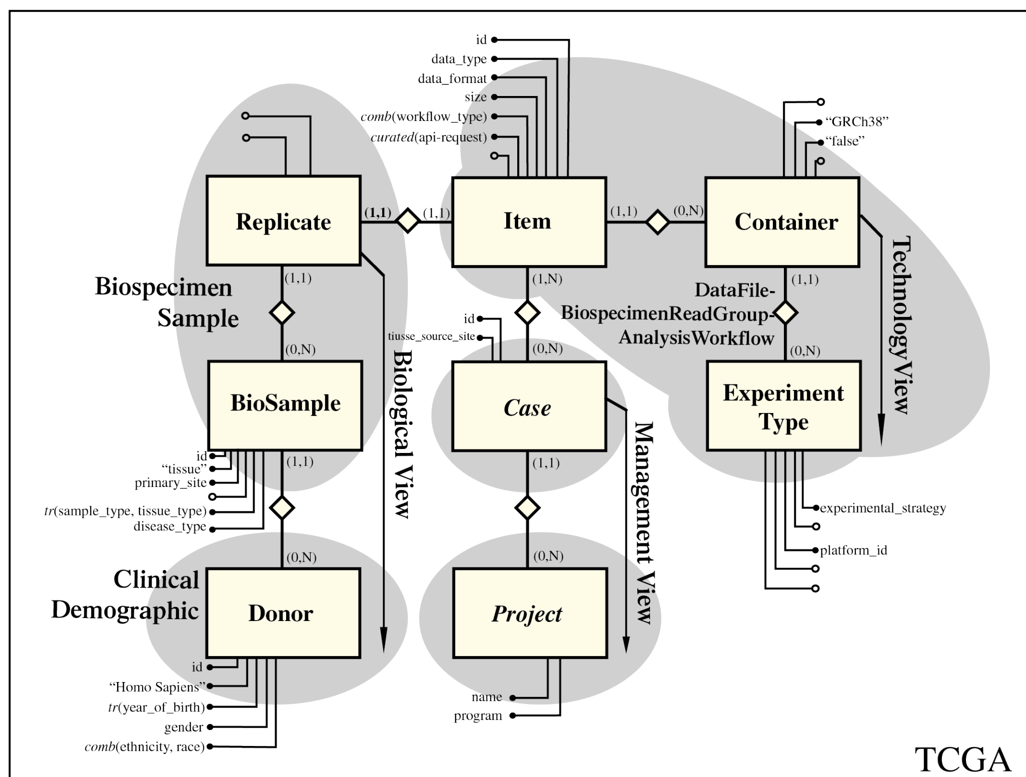
Vista di gestione

Questa vista consiste nella catena di entità **Item** - **Case** - **Project** e descrive il processo organizzativo volto alla produzione di ogni **Item** e il metodo con cui gli elementi sono raggruppati per la formazione di un **Case**: l'entità **Case** rappresenta una raccolta di **Item** specifici per un obiettivo di ricerca.
Project rappresenta il programma o il progetto svoltosi in un dato istituto (laboratorio o consorzio), che è responsabile della produzione dell'Item. *ProjectName* e *ProgramName* possono essere presenti ma non sono strettamente obbligatori.

3.3.2 Verifica delle sorgenti sullo schema GCM

Attraverso l'immagine 3.2, si intende dimostrare che la vista Globale (Entità centrale con vista biologica, tecnica e di gestione) risponde alla struttura e alle esigenze delle sorgenti prese in considerazione.
Negli schemi sono state utilizzate le seguenti notazioni:

- Gli attributi di ogni sorgente sono nella stessa posizione del GCM, ma sono stati utilizzati i nomi trovati nella documentazione di ogni sorgente, inoltre, attributi mancanti sono stati contrassegnati con cerchi bianchi.
- Le entità corrispondenti ad un singolo concetto nella sorgente di origine sono state "clusterizzate" utilizzando un'ombra grigia. Il nome dell'entità, in corsivo, corrisponde a quello originale della sorgente in esame e viene inserito all'interno dell'entità quando esiste una corrispondenza diretta fra l'entità del nostro schema globale e l'entità all'interno del repository considerato.
- Gli attributi gestiti manualmente sono stati racchiusi fra virgolette: inoltre, sono state utilizzate le notazioni di funzioni *tr, comb* e *curated* per descrivere rispettivamente la trasformazione di un campo sorgente, la combinazione di molteplici campi sorgente e campi gestiti manualmente.



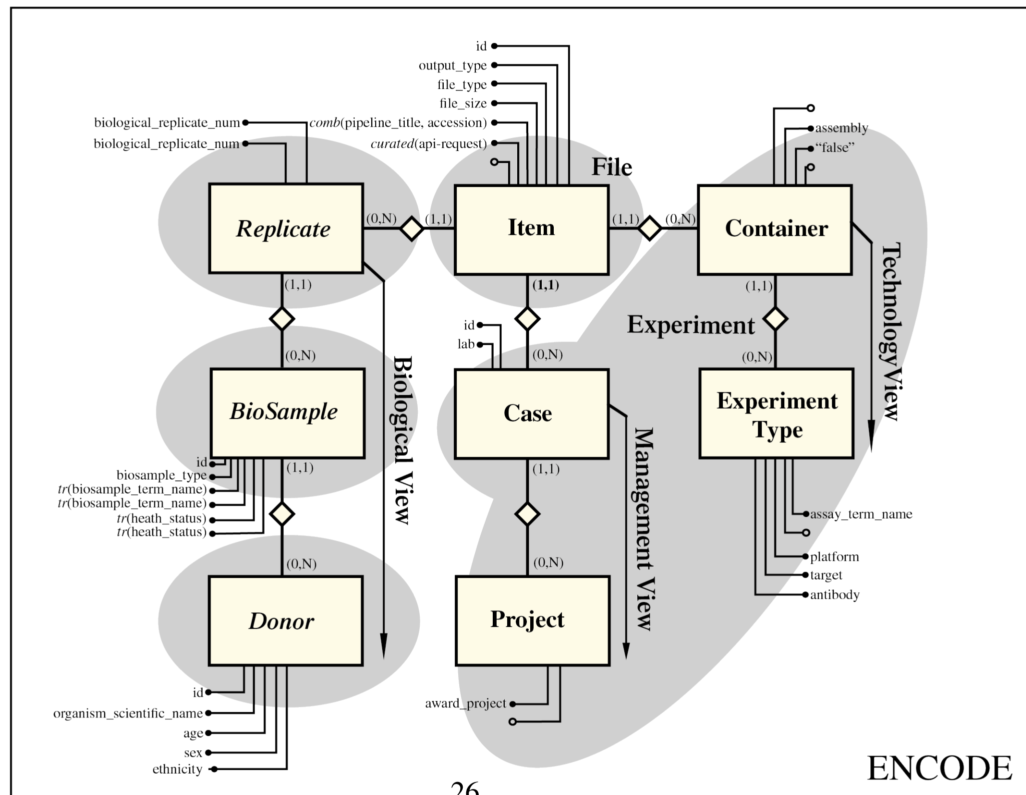


Figura 3.2: Confronto tra il GCM e lo schema delle sorgenti.

Capitolo 4

Implementazione del modello concettuale

Lo schema concettuale precedentemente presentato, seppur idoneo a descrivere un modello ideale, non è in grado di esprimere alcune peculiarità dei repository presi in esame.
Per questa ragione, è risultato necessario apportare al modello alcune modifiche che saranno elencate e descritte in questo capitolo. Prima di procedere con l'enumerazione, si intende sottolineare come queste modifiche siano risultate essenziali per alcune singolarità di specifici repository (specialmente ENCODE) e non per caratteristiche generiche dei dati genomici: esse assecondano, quindi, il funzionamento pratico dell'applicazione.

4.1 Definizione di chiave univoche per le tabelle

Le chiavi primarie delle tabelle, per motivi di efficienza delle operazioni di *join*, sono degli interi autoincrementali. Questa scelta, se da un lato rende molto efficiente il database in caso di query su più tabelle, dall'altro introduce un problema di gestione del dato: infatti, non è possibile utilizzare il valore intero per verificare se un dato elemento sia già presente all'interno del database, in quanto non portatore di alcuna semantica effettiva.
Da questa constatazione deriva la decisione di definire degli attributi univoci all'interno del database che permettano la corretta gestione della base di dati. Per le tabelle che definiscono un unico attributo come chiave univoca questo valore viene semplicemente controllato al momento dell'inserimento/aggiornamento della tupla. Al contrario, per le tabelle con molteplici valori di chiave univoca, viene considerata come chiave l'unione di tutti gli attributi: questo significa che una tupla, per essere considerata valida a livello di inserimento, deve avere non nullo almeno uno di questi attributi. Due tuple vengono considerate uguali quando tutti gli attributi che costituiscono la chiave univoca coincidono singolarmente.
In accordo con le definizioni contenute in [3] e con la documentazione dei vari repository, sono state effettuate le scelte elencate nella tabella 4.1. Questi attributi sono stati indicati come UNIQUE all'interno della base di dati, in modo da impedire anche a livello manuale l'inserimento di dati non conformi.

Tabella 4.1: Chiave univoche per tabella

Tabella	Chiave Univoca
Donor	*SourceId*
BioSample	*SourceId*
Replicate	*SourceId*
ExperimentType	*Technique + Feature + Target*
Project	*ProjectName*
Case	*SourceId*
Container	*Name*
Item	*SourceId*

Come si può verificare dalle tabelle di mapping, presentante in Appendice A e B per i due repository analizzati in questo lavoro di tesi, i singoli attributi possono anche essere frutto di molteplici valori a cui è stata applicata una funzione di trasformazione.

4.2 Ridefinizione del modello

Durante lo svolgimento del lavoro, si è palesata la necessità di apportare alcune modifiche al modello concettuale di partenza descritto in [3]. Tra questi:

1 Ricollocazione dell'attributo *Platform* dall'entità **Experiment Type** a **Item**;

2 Modifica della cardinalità tra le entità **Replicate** e **Item** da *1-N* a *N-N*;

3 Introduzione dell'attributo *ExternalRef* nell'entità **Case**;

4 Introduzione della relazione tra tuple dell'entità **Item** e altre tuple della stessa;

5 Ricollocazione dell'entità **Container** fra le entità *Case* e **Project**;

Queste modifiche sono rappresentate nell'immagine 4.1.
Tutte queste modifiche sono state effettuate per garantire la corretta impostazione della struttura e dell'organizzazione dei dati nel repository ENCODE. Per comprendere al meglio queste scelte, è opportuno considerare la struttura di un esperimento all'interno del repository.
Come si può notare dalla struttura di un esperimento ENCODE rappresentata nell'immagine 5.1, i formati di file contenuti all'interno di un esperimento "tipo" si suddividono in tre categorie di file:

- **fastq**: sono i file *raw* ricavati dalla sequenziazione del replicato tecnico sfruttando una determinata piattaforma (*platform*). Questi file (*item*) non presentano informazioni sulla *pipeline* utilizzata per ottenerli, essendo i primi elementi della sequenza di produzione dei dati. Con questo formato il valore di *dataType* assume sempre valore **reads**.

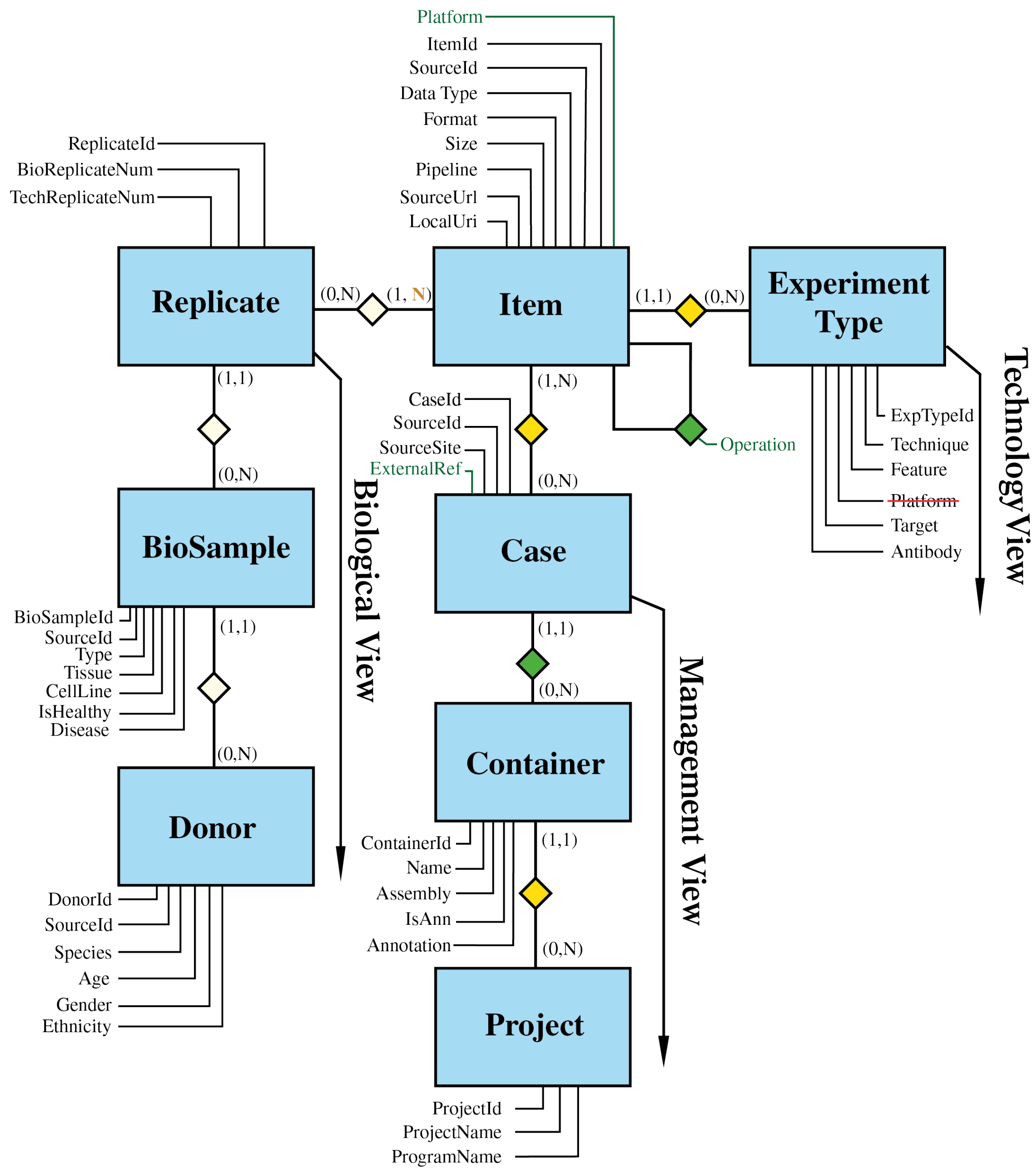


Figura 4.1: Diagramma ER del GCM modificato: sono indicate con in colore giallo le modifiche e in colore verde le aggiunte

- **bam**: sono i file intermedi ricavati dall'elaborazione dei file fastq o da altri file bam attraverso una determinata tecnica di pipeline. Questi file non presentano l'attributo platform, essendo delle elaborazioni di informazioni preesistenti. Con questo formato, il valore di *dataType* assume sempre valore **alignments**.
- **bed**: sono il prodotto finale della sequenza di produzione dei file di dati di un esperimento, ottenuti dell'elaborazione dei file bam. Questi file possono presentare sia l'attributo pipeline sia l'attributo platform. Con questo formato, il valore di *dataType* assume sempre valore **peaks**.

Nel repository integrato si memorizzano le informazioni (metadati) che documentano i file sopra descritti. In aggiunta a quanto proposto in [3], in cui sono considerati solo metadati contenuti in file di tipo **bed** (dati processati), si è deciso di inserire all'interno del repository integrato anche le informazioni riguardanti i file non processati, quindi anche i **fastq** e i **bam**, oltre che i **bed**.

La ricollocazione dell'attributo *Platform* risponde al bisogno di conferire la caratteristica al singolo item e non all'esperimento in generale.

La modifica della cardinalità fra la tabella **Replicate** ed **Item** è dovuta al fatto che, da un'analisi approfondita dei metadati di una fonte utilizzata durante il processo di integrazione (Encode), si è verificato che in alcuni casi un item deriva dal processamento di item che provengono da replicati differenti.

L'aggiunta dell'attributo *ExternalRef* è stata effettuata per poter tenere traccia della dipendenza fra vari esperimenti effettuati in repository differenti.

L'introduzione dell'entità **Derived From** si è resa necessaria per la memorizzazione delle dipendenze fra i vari Item appartenenti al medesimo esperimento.

Infine, la ricollocazione dell'entità **Container** si è palesata a seguito di un'analisi dettagliata dei dati ottenuti da una prima realizzazione del framework. Infatti, si è notato come generalmente un container non fosse relazionato solamente ad item di un unico ExperimentType, ma nello stesso tempo anche come l'ExperimentType fosse sempre unico per ogni item. Si è deciso, quindi, di spostare l'entità Container nella vista di Gestione, sfruttando la cardinalità *molti-a-molti* dell'entità **Case**, al fine di mantenere immutate le relazioni precedentemente elencate. Case è legato a *Container* con una relazione *uno-a-uno* e ad un Container corrisponde sempre e solo un **Project**: al contrario Project è collegato a Container con una relazione *zero-a-molti*; infine, Container è a sua volta legato all'entità Case con la medesima cardinalità.

4.3 Variazione delle proprietà degli attributi

Rispetto alle proprietà presentate nella Sezione 3.3.1, sono state apportate leggere modifiche alle caratteristiche degli attributi delle varie tabelle.

Tabella 4.2: Elenco delle proprietà degli attributi

Livello	Simbolo	Proprietà	Default
Sorgente	C	Contestuale	Non Contestuale
	D	Dipendente	Indipendente
	R	Ristretto	Libero
	M	Obbligatorio	Opzionale
Repository Integrato	H	Manuale	Estratto
	O	Ontologico	Ordinario
	T	Trasformazione	Nessuna operazione

Innanzitutto, poiché in un database classico relazionale non è possibile definire praticamente un attributo come multivalore, se non attraverso la creazione di una struttura apposita di supporto, è stata rimossa la proprietà di molteplicità di un attributo, che è stata gestita tramite la concatenazione di valori. Contrariamente, è stata aggiunta una nuova proprietà, **Trasformazione**, il cui inserimento esprime la necessità di modificare il tipo di un attributo in ingresso rispetto al tipo originale presente nella sorgente. Questa caratteristica si è rivelata utile in particolar modo per il casting degli attributi stringa nel tipo predisposto per la memorizzazione a livello di database, al fine di permettere il corretto svolgimento delle operazioni di popolazione.
Le proprietà utilizzate, comprensive delle nuove definizioni, sono elencate nella tabella 4.2. Nell'Appendice D sono riportate le tabelle di confronto tra le proprietà elencate nella sezione precedente e quelle realmente definite nell'applicativo, elencando inoltre il tipo del dato.

La proprietà di *Ristrettezza* è stata verificata empiricamente rispetto a dati attesi e disponibili a priori: da un'analisi preliminare, non sono state riscontrate disuniformità dei dati. Gli attributi che disponevano della proprietà di *Ristrettezza* a livello concettuale che non è stata poi prevista a livello di applicativo, suggeriscono che non si è ancora riusciti a definire un set determinato di valori a livello globale, oppure non si è ancora compreso se la proprietà esista realmente per quegli attributi. Come si può notare, molte proprietà di Contestualità o Dipendenza non sono presenti a livello applicativo perché le proprietà definite devono valere indipendentemente dalla sorgente e quindi a livello globale: anche se si fosse in grado di definire delle regole plausibili, ad ogni modo manca uno strumento che permetta di ottenere tutte le ontologie a partire da un singolo valore: questa disuniformità di rappresentazioni dei valori determinerebbe la necessità di istituire un set di regole troppo ampio per poter trattare tutti i casi. Si auspica invece che, una volta predisposte le ontologie dei valori, sia possibile definire delle regole generali che non vengano alterate dal variare delle sorgenti supportate e in cui non si presenti la necessità di implementare nuovamente tutte le dipendenze ogni qual volta si definisce una nuova sorgente.
La proprietà *Manuale* viene applicata a livello di repository e non a livello di schema globale: tuttavia, esistono casi in cui essa potrebbe essere applicata (si pensi al nome del progetto per l'entità Project).

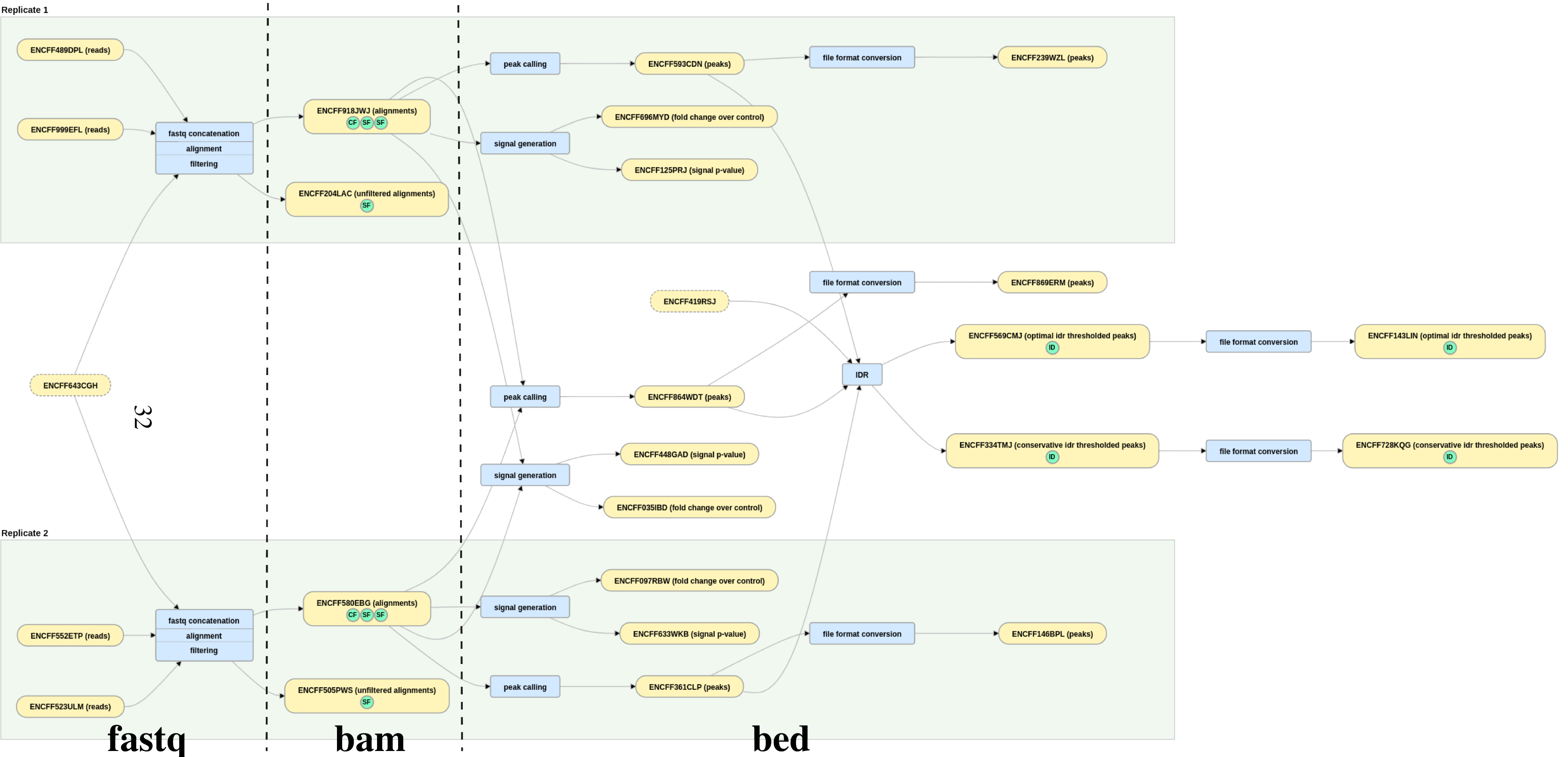


Figura 4.2: Encode Experiment con le etichette gialle vengono indicati i file, con le etichette azzurre vengono indicate le tipologie di operazioni attuate per la loro elaborazione in un nuovo file processato.

4.4 Revisione delle regole e dipendenze

A livello pratico, le regole che esprimono **dipendenza di esistenza** (o *Contestualità*) e **dipendenza del valore** (o semplicemente *Dipendenza*) possono essere effettivamente implementate oppure verificate. Con il termine "implementate" si intendono regole la cui validità è certa, in quanto il codice è predisposto al loro supporto: a valori in input dalla sorgente possono essere applicate regole che ne modificano il valore in conformità alla dipendenza desiderata.
Le regole sono state suddivise in due categorie:

- Intratabellari: interessano colonne di un'unica tabella.
- Intertabellari: interessano colonne di più tabelle.

La semplice verifica delle regole (e non quindi la loro effettiva implementazione) può essere svolta secondo due modalità:

- *debole*: la loro violazione è verificata a posteriori della popolazione del modello interno al framework e, nel caso queste risultassero violate, un operazione di log ne terrebbe traccia nel report delle statistiche; indipendentemente dal fatto che la violazione sia avvenuta, le tuple sono comunque caricate nel repository condiviso.
- *forte*: eventuali tuple che non rispettano le regole non sono inserite nel database e un messaggio di log ne tiene traccia durante l'esecuzione del framework.

Si è scelto, per il momento, di verificare le regole esclusivamente nella modalità di tipo *debole*: questa operazione, infatti, consente di effettuare un'analisi preliminare dei metadati e del loro comportamento senza impedirne la registrazione nel database. Un valore booleano nell'*application configuration file* (vedi Sezione 6.11) del framework permette di modificare, nel caso si ritenesse opportuno in futuro, la modalità del controllo delle regole, trasformandola in *forte*.

Nel prototipo attuale dell'applicativo sono state implementate le regole di *Contestualità* che specificano la non esistenza del valore nel caso di regole intratabellari; una di queste regole, ad esempio, vincola i casi in cui il valore dell'attributo *tissue* della tabella BioSample deve essere nullo. Nel dettaglio, le regole sono espresse nella tabella 4.3.
Regole di dipendenza di esistenza obbligatoria 4.4, sia intertabellari che intratabellari, non possono essere implementate attivamente, in quanto per ogni sorgente di riferimento non si può essere certi che un valore esista a fronte di un altro: si tratta, infatti, di una caratteristica intrinseca del repository di riferimento, per la quale non possediamo sufficienti informazioni che possano verificarla. Il problema consiste nel trovare valori di Default validi per ogni sorgente nel caso in cui il valore scelto dai dati in ingresso dalla sorgente, per mezzo di regole di mappatura, non esista.
Regole che specificano Dipendenza di Valore sia intratabellari (vedi Tabella 4.5) che intertabellari (per ora non esistono esempi sviluppati nel software) sono anch'esse solamente loggate, per il momento, per fini statistici. In caso si volesse modificare il loro comportamento da semplice verifica ad effettiva implementazione applicando la regola pre inserimento/aggiornamento,

Tabella 4.3: Regole di dipendenza intratabellari

Tabella	Attributo	Regola
Container	*annoation*	*isAnn* = false → Null
Biosample	*disease*	*isHealthy* = true→ Null
Biosample	*tissue*	*type* = "cellLine"→ Null
Biosample	*cellLine*	*type* = "tissue"→ Null
Biosample	*disease*	*isHealty* = false → Null

Tabella 4.4: Regole di obbligatorietà intratabellari

Tabella	Attributo	Regola
Biosample	*tissue*	*type* = "tissue" → Not Null
Biosample	*cellLine*	*type* = "cellLine" → Not Null
Biosample	*disease*	*isHealthy* = true → Not Null
Item	*platform*	[*format* = "fastq" → Not Null OR dataType = "reads" → Not Null]
Item	*pipeline*	*format* = "bam" → Not Null
ExperimentType	*target*	*Technique* = "Chip-seq" → Not Null
ExperimentType	*antibody*	*Technique* = "Chip-seq" → Not Null
Container	*annotation*	*isAnn* = true → Not Null

bisognerebbe solo tener presente di eventuali dipendenze circolari. Per ora si è preferito applicare esclusivamente una verifica di queste regole e non la loro implementazione in quanto, oltre a non aver a disposizione tutte le regole, per ottenerle sarebbe necessario uno studio più dettagliato sui singoli valori di tutte le sorgenti che si vorranno supportare; inoltre, sarebbe indispensabile una struttura di supporto che fornisca i sinonimi dei valori contenuti nelle regole in tutte le sorgenti supportate, funzione che sarà introdotta con il supporto alle ontologie. Infine, un ulteriore problema consiste nel selezionare un valore fra i molteplici mappati dalla dipendenza, come nel caso mostrato nella tabella 4.5.
È possibile, come è stato svolto per le regole di Contestualità descritte nella tabella 4.6, che certe regole vengano definite solo a fini statistici utili a verificare, confermare e validare supposizioni sui dati ottenute in ingresso da determinate sorgenti.
La Tabella 4.7 riassume la situazione attuale riguardante l'implementazione o la verifica delle regole di Contestualità e Dipendenza di valore all'interno del framework.

Tabella	Attributo	Regola
Container	assembly	Donor.species="Homo Sapiens" ->["hg19", "GRCh38"]

Tabella 4.5: Regole di dipendenza di Valore.

Tabella	Attributo	Regola
Item	*platform*	format = fastq → Null
Item	*platform*	dataType = reads → Null
Item	*pipeline*	format = bam → ->Null

Tabella 4.6: Regole utilizzate per la verifica di proprietà nel contesto di ENCODE.

Tabella 4.7: Panoramica attuale dello sviluppo delle regole di Constestualità e Dipendenza nel framework

Tipologia di dipendenza	**Dipendenze di Esistenza**		**Dipendenze di Valore**
	Di esistenza	Di non Esistenza	
Intratabellare	Verificate (debole)	Implementate	Verificate (debole)
Intertabellare	Verificate (debole)	ND	Debole

4.5 Introduzione di funzioni per la trasformazione dei dati in input

Al fine di discriminare particolari situazioni in cui il dato in ingresso non si presenta nel formato o nel tipo desiderato, sono state applicate funzioni di trasformazione per un numero limitato di attributi. Le funzioni di trasformazione si dividono in due tipi:

- **Trasformazione del tipo**: è una funzione che, dato il parametro stringa, ottenuto in ingresso dal mapping, trasforma il dato nel tipo dell'attributo descritto dal modello del sistema. Funzioni di trasformazione del **tipo** sono ad esempio quelle per gli attributi *bioReplicateNum* e *techReplicateNum* dell'entità **BioSample**, sia per ENCODE che per TCGA, dove il valore estratto dai dati in ingresso dalle sorgenti è castato da stringa ad intero.

- **Trasformazione del dato**: è una funzione che, dato il parametro in ingresso dal mapping definito dall'utente, applica uno statement di tipo *switch-case* che restituisce un valore contenuto in un set prestabilito. Come si può notare, contrariamente alle regole presentate nella sezione precedente, questa dipendenza non interessa colonne di una o più tabelle, ma dipende esclusivamente dal parametro in ingresso definito dalla configurazione dell'utente. In questi casi, una funzione che implementa la dipendenza si preoccuperà di assegnare all'attributo il valore corretto. Funzioni di trasformazione del dato sono rappresentate, ad esempio, dall'attributo *feature* per l'entità **Donor** in TCGA.

Si noti che le regole di Trasformazione del tipo sono generiche e definite a livello di modello e non a livello di sorgente: risultano quindi identiche per ogni repository considerato. Esistono casi in cui le due trasformazioni sono utilizzate sul medesimo attributo. Un esempio esplicativo è fornito dall'attributo *age* che rappresenta l'età, espressa in giorni, del donatore al momento della donazione del campione biologico appartenente all'entità **Donor**. In entrambi i repository ENCODE e TCGA, nei dati in ingresso dalle sorgenti, non era inizialmente presente il dato nel

formato desiderato: in particolare, nel caso di TCGA il dato era espresso con un valore negativo, mentre in ENCODE il dato poteva essere riportato in unità di misura variabili (anni, mesi, settimane o giorni), espressi sia in singolo valore ma potenzialmente anche usando un range di valori; inoltre, una funzione di casting del valore da stringa ad intero è in entrambi i casi necessaria.
Una particolare funzione di trasformazione del dato è applicata per l'attributo *sourceSite* dell'entità **Case** per l'import dei dati dalla sorgente TCGA. Esclusivamente per questa situazione caratteristica, viene applicata una funzione di mapping del parametro in ingresso con valori contenuti in una tabella di supporto del database dedicata unicamente a questo scopo. Non vi è alcun legame di dipendenza di chiavi fra queste tabelle, ma una query si incarica di recuperare il valore dal database ed assegnarlo all'attributo. La motivazione che ha portato ad utilizzare questa tabella è la grande mole di possibili valori mappati: inoltre, i valori sono stati recuperati sul web[1] da una tabella statica e quindi difficilmente modificabile nel breve periodo ma facilmente importabile in caso di necessità.
Tutte le regole di trasformazione per il repository ENCODE sono elencante nella Tabella 4.8 mentre quelle per TCGA sono elencate in 4.9.

Tabella 4.8: Funzione per repository ENCODE

Tabella	**Attributo**	**Funzione**	**Di tipo**	**Di dato**
Biosample	*isHealthy*	if(param.contains("healthy")) true else false	Sì	Sì
Donor	*age*	case "year" → param.toInt * 365 case "month" → param.toInt * 30 case "day" → param.toInt	Sì	Sì
Item	*size*	param.toLong	Sì	No
Replicate	*bioReplicateNum*	param.toInt	Sì	No
Replicate	techReplicateNum	param.toInt	Sì	No
Container	*isAnn*	if((param).equals("true")) true else false	Sì	No

[1] https://gdc.cancer.gov/resources-tcga-users/tcga-code-tables/tissue-source-site-codes

Tabella 4.9: Funzioni di trasformazione GDC

Tabella	Attributo	Funzione	Di tipo	Di dato

Biosample	*isHealthy*	param match {
		case "Additional - New Primary" $\rightarrow$ false;
		case "Primary Tumor" $\rightarrow$ false;
		case "Solid Tiusse Normal" $\rightarrow$ true;
		case "Metastatic" $\rightarrow$ false;
		case "Blood Derived Normal" $\rightarrow$ true
		}
			Sì	Sì

Donor	*age*	abs(param.toInt)	Sì	Sì

Item	*feature*	param match{
		case "RNA-seq" $\rightarrow$ "gene expression",
		case "DNA-seq" $\rightarrow$ "mutations",
		case "miRNA-Seq" $\rightarrow$ "gene expression",
		case "Methylation Array" $\rightarrow$ "DNA-methylation",
		case "Genotyping Array" $\rightarrow$ "Copy Number Variation"
		}
			No	Sì

Item	*size*	param.toLong	Sì	No
Case	*sourceSite*	getSourceSite(param)	No	Sì
Replicate	*bioReplicateNum*	param.toInt	Sì	No
Replicate	techReplicateNum	param.toInt	Sì	No
Container	*isAnn*	if((param).equals("true")) true else false	Sì	No

Capitolo 5

Architettura

In questo capitolo, saranno elencate e giustificate le scelte architetturali e le scelte tecnologiche attuate per lo svolgimento del lavoro.

5.1 Utilizzo del linguaggio Scala

Le motivazioni che hanno portato ad un'implementazione del codice, piuttosto che all'utilizzo di tool di integrazione dati, sono molteplici. La principale risiede nell'intenzione di mantenere una coerenza rispetto ai lavori precedentemente svolti: questa scelta permette, infatti, di instaurare una semplice integrazione con il software già presente e garantisce una continuità nello sviluppo.
In secondo luogo, si ritiene di primaria importanza sviluppare un'architettura più flessibile, dove in futuro sarà più semplice implementare, utilizzando lo schema già sviluppato, l'integrazione del software con altre sorgenti che presentino una struttura di presentazione e formattazione dei dati completamente diversa rispetto ad ENCODE e TCGA: questa opportunità è risultata essere una motivazione determinante per protendere alla scelta di un'implementazione del codice, che rende possibile ciò.
Un altro fattore considerato risiede nel voler garantire una più semplice gestione futura del codice, poiché, per poter effettuare modifiche al programma, sarà sufficiente la conoscenza di Scala[22]. Diversamente, nel caso in cui avessimo optato per l'utilizzo di un tool di integrazione, questo avrebbe implicato la necessità di conoscere uno strumento complesso, dotato di una curva di apprendimento molto alta, con conseguenti difficoltà nella modifica anche delle più semplici funzionalità nelle correzioni di eventuali bug.
Infine, l'implementazione di codice permette di condividere il software esternamente, attraverso strumenti come GitHub, rendendo così più agevole il lavoro di revisione e possibile contribuzione esterna da parte di altri utenti interessati al progetto.

5.2 Libreria Slick

Partendo dal modello dei dati presentato in [3], è stato definito il modello dei dati standard utilizzando la libreria Slick [27]. Slick è una moderna libreria di tipo Functional Relational Mapping

(FRM) di accesso e interrogazione di base di dati di tipo relazionale per Scala che permette di lavorare con dati memorizzati su database (stored data) trattandoli come se fossero collezioni Scala e offrendo, nello stesso istante, completo controllo sugli accessi al database e sul trasferimento dei dati.
È stato quindi possibile scrivere le query del database in Scala anziché SQL, sfruttando vantaggiose caratteristiche del linguaggio di programmazione: tra queste, il controllo statico di entità e query, la sicurezza, a tempo di compilazione, rispetto ai tipi e la "composizionalità" di Scala tipica dei linguaggi funzionali. Slick presenta un compilatore di query estensibile che può generare codici per diversi backend: in questo modo, il framework è reso indipendente dalla tecnologia utilizzata, con la possibilità di concentrarsi unicamente sulla logica della gestione dei dati.
In altri termini, l'utilizzo di Slick ha permesso di impiegare un unico linguaggio per l'implementazione del codice, sfruttando la logica e la duttilità della programmazione orientata agli oggetti per la gestione delle interazioni con il database.

5.3 Struttura generale del programma

Il framework implementa un classico processo ETL (Sezione 2.5) per l'estrazione, la trasformazione il caricamento dei dati sul repository condiviso. I dati non sono scaricati direttamente da una sorgente web, ma sono ottenuti tramite API dalla sorgente e salvati localmente sulla macchina: solo successivamente verranno analizzati. Questa scelta, se pur dispendiosa in termini di spazio su disco, permette di utilizzare gli stessi file anche per altri scopi non direttamente collegati a questa tesi, ma utili nel contesto del progetto GeCo. Inoltre, i file utilizzati, che in seguito chiameremo *data file* o file dei dati, per i repository ENCODE e TCGA non sono gli originali scaricati dalle sorgenti, ma risultano già elaborati sfruttando tool preesistenti descritti nella Sezione 5.4. I file analizzabili, come di consueto nei processi ETL, possono presentarsi in diversi formati: per ora sono supportati dati in formato attributo-valore di tipo tab-delimited. Diversamente dai classici modelli ETL, la logica di trasformazione e inserimento dei dati non è descritta esclusivamente nella *business logic* del framework, ma il suo comportamento dipende anche da un file di configurazione dato in input assieme al *data file* (Sezione 6.4). Questo permette di modificare il comportamento del framework in base alle diverse sorgenti, massimizzare il riutilizzo di codice e permettere di variare comportamenti senza l'obbligo di modificare il codice sorgente. Per poter sviluppare il processo di trasformazione, è stato definito, per mezzo di classi Scala, il modello del GCM (Sezione 5.5). In queste istanze, gli attributi sono modificati ed elaborati sia per mezzo di informazioni contenute nel file di configurazione (Sezione 6.4.2), che per mezzo di regole di dipendenze (Sezione 4.4) implementate a livello di codice: questo significa che i dati caricati non sono una semplice copia dei dati in input dalle sorgenti, ma subiscono una trasformazione sia per essere conformi allo schema logico preposto, ma anche per essere maggiormente gestibili e analizzabili dall'utente finale. Una volta elaborati i dati, si prosegue alla fase di caricamento dei dati sul repository condiviso. Questo processo utilizza la libreria Slick (Sezione 5.2) per interfacciarsi con la base di dati. Le operazioni sono svolte tutte in modo sequenziale per ogni data file analizzato, al quale possono essere correlati più inserimenti o aggiornamenti per la stessa tabella. Gli unici dati del modello reperiti durante la fase

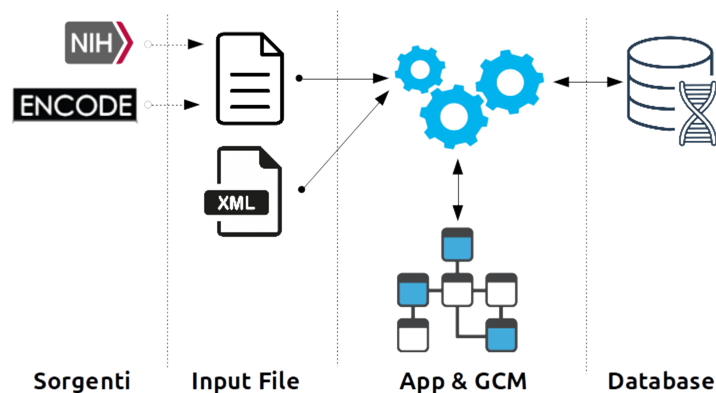


Figura 5.1: Rappresentazione ad alto livello del framework.

di inserimento sono i riferimenti alle chiavi esterne delle tabelle correlate. Un grande vantaggio introdotto dal processo ETL risiede nella possibilità di verificare, elaborare e validare i dati in input, rispetto a specifiche indicate a priori: in questo modo c'è possibilità di trarre conclusioni sui dati in ingresso utilizzando un approccio di tipi *data-informed*[1].
Una delle specifiche fondamentali del framework prevedeva che futuri inserimenti di dati da nuovi repository fossero facilmente integrabili nell'architettura sviluppata e che cambiamenti dei mapping tra attributi delle sorgenti e attributi del modello concettuale non necessitassero di alcun cambiamento nel codice sorgente. Queste specifiche hanno portato alla definizione del funzionamento generale del programma illustrato nella Figura 5.1.
Infine, tramite una procedura di esportazione dei dati, è possibile estrapolare le informazioni in un file di tipo TSV partendo dal singolo item e procedendo alle entità esterne dello schema del modello GCM.

5.3.1 Indipendenza dei processi ETL

Aspetto fondamentale del framework è la distinzione, sfruttando la struttura tipica dei processi ETL descritta nella Sezione 2.5, delle tre sezioni che compongono il software. Queste tre sezioni (estrazione, trasformazione e caricamento dei dati) sono del tutto indipendenti fra loro e ognuna mostra a quella precedente un'interfaccia che deve essere rispettata (ad esempio un metodo che accetta una determinata struttura del dato che deve essere passato alla fase successiva) in modo tale che ognuna delle fasi possa operare anche a fronte della modifica della fase precedente o successiva ad essa, a garanzia del rispetto dell'interfaccia. Il disaccoppiamento delle funzionalità delle operazioni permette una struttura più estendibile del codice grazie allo sviluppo di nuovi moduli indipendenti che possono essere integrati in una delle tre fasi, in maniera del tutto trasparente al codice già sviluppato. Inoltre, future necessità di modifica di parti già sviluppate,

[1]Un approccio *data-informed* è simile ai classico approccio *data-driven*, ma diversamente da quest'ultimo nel primo l'esperienza e la comprensione delle informazioni giocano un ruolo importante nelle tue decisioni come l'informazione stessa

per modifiche delle esigenze e delle specifiche del software, non hanno alcun impatto sulle altre parti del programma.

5.4 Utilizzo di tool preesistenti

Durante lo svolgimento della tesi, è stato sfruttato un tool sviluppato precedentemente nel contesto del progetto GMQL e già presentato nella tesi di laurea magistrale intitolata *Automation of retrieval, transformation and uploading of genomic data and their metadata for their integration into a GDM repository* [29]. Questo software verrà chiamato d'ora in poi, per comodità, *Downloader and Transformer*.
Uno dei compiti del software è quello di scaricare sia dati che metadati provenienti da diversi repository di dati genomici pubblici: in particolare, dal portale ENCODE, il software scarica i file di dati processati *bed* e i contestuali file *json* che ne rappresentano i metadati. Tramite un algoritmo che appiattisce la struttura ad albero originale ed esclude informazioni non utili dal file json (che si riferisce all'intero esperimento genomico), il software si incarica di produrre un nuovo file testuale in formato ⟨chiave, valore⟩ che riassume tutti i metadati che si riferiscono solamente allo specifico file in formato *bed*. Le chiavi corrispondono al path del file *json* in input, mentre il valore rispecchia l'attributo dell'elemento json. Quest'ultimo è il file utilizzato in input al programma per le operazioni di mapping descritte nell'Appendice A per le importazioni dei dati dalla sorgente ENCODE.
Se da un lato il file *bed.meta.json* ha offerto un grandissimo aiuto nelle operazioni di analisi e di mappatura degli attributi del GCM, dall'altro soffre della mancanza di informazioni dei file da cui l'Item *bed* deriva. Questa mancanza è giustificata dal fatto che, nelle intenzioni iniziali, non è stato contemplato l'inserimento dei file non processati all'interno del database, ma questa specifica è stata successivamente modificata in fase di realizzazione del software per ragioni di completezza, considerando che gli attributi *Platform* e *Pipeline* sono caratteristici, solamente in ENCODE, dei dati non processati e non del file *bed*.
Si è voluto mantenere l'utilizzo del file con gli attributi appiattiti e utilzzare il file *json* solo come supporto, per facilità di lettura da parte dell'utente in caso si necessitasse di modifiche delle mappature fra gli attributi del repository sorgente e del GCM: questo file risulta infatti di rapida interpretazione essendo privo dei livelli tipici dei file json ed essendo molto contenuto in termine di righe.

Un altro tool di trasformazione dei dati impiegato in questa tesi è TCGA2BED [10], un software che, fra le molteplici funzionalità, permette di trasformare i dati provenienti dalla sorgente TCGA in un formato di tipo ⟨chiave, valore⟨ del tutto simile a quello ottenuto dal *Downloader and Transformer*. TCGA2BED è un software scritto in Java che permette l'estrazione, l'estensione e l'integrazione di dati genomici, metadati clinici e campioni biologici (biospecimen) provenienti da TCGA e trasformati rispettivamente in file di formato *bed* oltre che in file ⟨attributo,valore⟨ tab-delimited. Inoltre, supporta la conversione dei file genomici in CSV, GTF, JSON e XML. Il software è disponibile come applicazione desktop e la sua architettura è basata sul pattern Model-View-Controller[2]. I file prodotti dal tool sono utilizzati come *data file* in input

[2]Model-view-controller (MVC, talvolta tradotto in italiano con la dicitura modello-vista-controllo) è un pattern

al programma per l'esecuzione del framework per l'importazione delle informazioni dalla sorgente TCGA, i cui dati saranno mappati nel database condiviso sfruttando le regole di mappatura descritte in Appendice A.

5.5 Architettura del codice e delle classi

Di seguito è riportata l'architettura implementata:

Le classi sviluppate possono essere distinte in tre macrocategorie:

- Gestore del Database
- Modello delle tabelle
- Raccoglitore di tabelle

5.5.1 Gestore del Database

Il gestore del database, o **DBHandler**, è una singola classe di tipo *singleton*[3], sviluppata sfruttando la libreria *Slick*, che coordina l'interazione fra il software e la base dati.
Slick, come precedentemente specificato, permette di gestire gli *stored data* come se fossero collezioni Scala, garantendo, quindi, tutte le funzionalità di una vera e propria classe.
Il DBHandler definisce la struttura di ogni singola tabella elencandone attributi, chiavi primarie, chiavi esterne, attributi univoci ed eventuali valori di default o obbligatorietà.
Una routine di inizio programma verifica la presenza delle tabelle sul database, le cui specifiche di accesso sono indicate nel file di configurazione e, nel caso queste non fossero presenti, le crea. Il controllo di presenza è un semplice confronto tra il nome specificato nel codice e i nomi delle tabelle presenti nel database considerato. È fondamentale che il processo di creazione delle tabelle sia effettuato dall'applicativo e non manualmente da un utente: in quest'ultimo caso, infatti, si potrebbe commettere l'errore di definire per le tabelle degli attributi che non rispecchiano le specifiche elencate nella loro definizione, creando problemi di compatibilità tra il codice sorgente e la base dati; quindi, è fondamentale che le caratteristiche degli attribuiti siano concordi.
Un esempio dimostrativo è rappresentato da un utente che, per l'attributo *size* della tabella Item, inserisce, come tipo di dato, una variabile intera causando al momento dell'inserimento dei dati una perdita di informazioni dovuta ai valori di dimensioni elevate presenti per il repository ENCODE mentre, al fine del corretto funzionamento del programma, sarebbe necessario immettere un tipo Long. Un'altra motivazione è la necessità di creare la struttura delle tabelle seguendo le caratteristiche di unicità e annullabilità di alcuni attributi, inserendo, nel caso essi non esistano, i corretti valori di default.
Non è possibile, sfruttando le funzioni di *Slick*, aggiornare dinamicamente la struttura di una

architetturale molto diffuso nello sviluppo di sistemi software, in particolare nell'ambito della programmazione orientata agli oggetti, in grado di separare la logica di presentazione dei dati dalla logica di business. Questo pattern si posiziona nel livello di presentazione in una architettura multi-tier.

[3]Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

tabella. Nel caso in cui un nuovo attributo fosse aggiunto o fosse necessario un qualsiasi cambiamento su una tabella, questa modifica comporterebbe l'obbligo di cancellazione totale della tabella in questione, al fine di permettere al framework di ricrearne la struttura completa: quindi, nel caso in cui non si volessero perdere i dati collezionati nel database, una semplice operazione manuale di *dump* dei dati della tabella interessata risulterebbe opportuna prima della sua cancellazione.
Oltre alla creazione della loro struttura, la classe **DBHandler** presenta, per ogni tabella, i metodi di inserimento, update e ricerca (per mezzo di specifici valori) delle tuple; essa fornisce, inoltre, un metodo di verifica per la presenza o la mancanza di una determinata tupla, data la chiave univoca di rappresentazione, presentata nella tabella 4.1 illustrata nella precedente Sezione.
La classe è stata definita come *Singleton* per risolvere eventuali problemi di concorrenza: infatti, sfruttando questo *Design Pattern*, non possono esistere diverse istanze della classe all'interno dell'applicazione, ma essa è sempre unica. Questo, a meno di molteplici istanze dell'applicazione stessa, risolve a priori il problema dell'accesso concorrente alla base di dati.
I metodi specificati sono indipendenti dalla versione di database utilizzato, in quanto nessuna riga di linguaggio SQL è presente all'interno del codice: dunque, nel caso in cui si volesse modificare la tipologia di database, l'unico parametro da modificare sarebbe il driver di comunicazione importato all'interno della classe.

5.5.2 Model delle tabelle

È il cuore dell'architettura sviluppata e si suddivide in quattro parti:

- La classe *Table*[4] di tipo Trait[5] specifica tutti i metodi che ogni tabella deve implementare per svolgere la propria funzione in modo corretto e fornisce gli attributi comuni a tutte le tabelle.
  Esempi di metodi specificati in questa classe sono: *insert()*, *update()* e *setForeignKey()*. Attributi comuni a tutte le tabelle sono la/le *primaryKey(s)* e le *foreignKeys*. Questi valori sono stati introdotti per ottenere la massima indipendenza fra le classi istanziate: infatti, le singole istanze non hanno visione di tutti gli attributi delle altre classi, ma solamente delle chiavi primarie e delle chiavi esterne, queste ultime sono salvate in duplice copia, sia nello specifico attributo sia in questi speciali campi forniti dalla *Table*.

- La classe Trait *<TableName>*[6] estende *Table* e definisce le specifiche di una singola tabella. È stata definita una classe Trait *<TableName>* per ogni tabella contenuta nel GCM e sono state inserite, inoltre, tutte le tabelle di congiunzione per risolvere le relazioni *molti a molti*.

[4]Sono indicate in *corsivo* le classi di tipo Trait, in ***corsivo-grassetto*** le classi astratte e in **grassetto** le classi istanziabili.

[5]Per chi non ha familiarità con il linguaggio Scala, può immaginare una classe *Trait* come una *Interface* nel linguaggio Java.

[6]Quando le classi vengono racchiuse utilizzando i simboli 'minore' e 'maggiore' (<>) non si intendono classi realmente implementante nel contesto del framework, ma semplicemente delle classi "logiche" a cui è definito un compito che deve essere portato a termine da un'altra classe definita all'interno dell'applicativo.

Sono state ottenute, quindi, molteplici classi Trait: *Donor*, *Biosample*, *Replicate*, *Case*, *Container*, *ExperimenType*, *Project*, ed *Item* (una per ciascuna tabella del GCM) oltre a *CaseItem*, *ReplicateItem* e *DerivedFrom* (per rappresentare le cosiddette "tabelle ponte" tra diverse entità): ognuna di esse elenca i singoli attributi della tabella di riferimento, specifica le dipendenze con altre tabelle e ne implementa i metodi propri, tra cui *insert()*, *update()*, *setForeignKeys()* ecc.

- Le vere e proprie classi istanziate sono definite attraverso un nome che deriva dal concatenamento del nome della <*TableName*> con il repository sorgente di riferimento. Un esempio di classe potrebbe essere **DonorEncode** nel caso in cui la classe in questione estenda la classe trait *Donor* e si riferisca alla procedura di import del repository ENCODE. Questa classe ha il compito di definire il metodo *setParameter()*, che specifica la mappatura dei valori in input attraverso il metodo di inserimento definito nel file di configurazione. In caso di necessità, la classe può sovrascrivere i metodi definiti in <*TableName*> con implementazioni proprie, con lo scopo di gestire situazioni particolari in singole sorgenti: questa funzionalità è stata impiegata, ad esempio, per la definizione della classe **ItemEncode** che ha ridefinito *insert()*, un metodo che ha lo scopo di inserire il record all'interno del database, definito in *Item*. In questa sorgente, vi era effettivamente la necessità di svolgere la routine di recupero degli item da cui l'elemento analizzato deriva, processo che non è presente genericamente per tutti i repository in quanto solo in Encode è definita questa particolare struttura per un esperimento.

- La classe astratta ***<RepositoryTable>*** è estesa da tutte le istanze della stessa sorgente: essa ha il compito di definire comportamenti condivisi dalle tabelle di una determinata sorgente e di consentire la condivisione dei dati fra le varie tabelle in caso di necessità, tramite l'utilizzo del pattern di *Dependecy Injection*[7].
  Quest'ultima feature è stata impiegata nell'implementazione delle classi per le tabelle di ENCODE: essendo infatti possibile che un singolo file in input possedesse informazioni di molteplici replicati, biosample, donatori e item, è risultato utile uno strumento che permettesse di relazionare correttamente le varie informazioni lette dal file in input, al fine di mettere in comunicazione le varie istanze delle classi. Il nome delle classi propriamente sviluppate nel contesto del framework è il risultato della concatenazione del repository sorgente, a cui la classe fa riferimento, con la stringa costante "Table": un esempio di nome della classe potrebbe essere, quindi, ***EncodeTable***.

Nell'immagine 5.2 è presentato un estratto di diagramma UML che espone le relazioni che intercorrono fra le classi appena citate.

[7]Dependency injection (DI) è un design pattern della Programmazione orientata agli oggetti il cui scopo è quello di semplificare lo sviluppo e migliorare la testabilità di software di grandi dimensioni. Per utilizzare tale design pattern, è sufficiente dichiarare le dipendenze di cui un componente necessita (dette anche interface contracts). Quando il componente verrà istanziato, un iniettore si prenderà carico di risolvere le dipendenze (attuando dunque l'inversione del controllo).

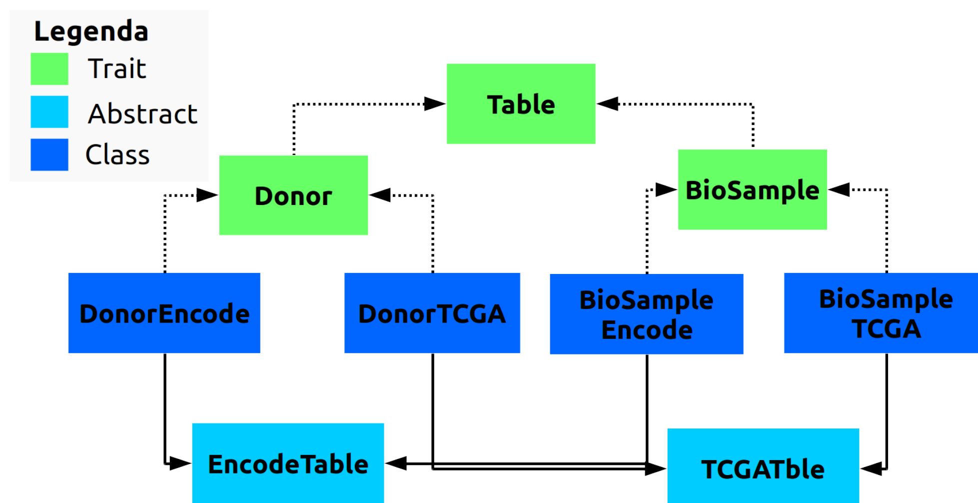


Figura 5.2: Estratto di diagramma UML Model delle tabelle.

5.5.3 Raccoglitore di tabelle

Il raccoglitore di tabelle è la struttura che ne "colleziona" tutte le istanze e ne permette il coordinamento. Facendo da ponte con la classe **Main**, gestisce le operazioni di inserimento, aggiornamento, settaggio delle chiavi esterne e verifica la correttezza dei dati in ingresso. Questa tabella ha una visione d'insieme di tutta l'esecuzione del programma e consente di localizzare in un unico punto tutte le procedure di gestione che non dipendono dalla singola istanza di una tabella.
Il raccoglitore è a sua volta suddiviso in due livelli di astrazione:

- La classe *Tables* di tipo Trait è l'interfaccia che specifica tutte le tabelle presenti all'interno del repository condiviso e fornisce il corretto ordine di inserimento delle tuple all'interno del database, con la loro relativa routine.
  Questa classe estende una *Enumeration*, in modo da poter collegare univocamente uno specifico valore ad ogni tabella. Con questo sistema, è stato possibile mettere in relazione le singole istanze ai valori stringa che per comodità corrispondono agli stessi nomi delle tabelle. Questa relazione è molto utile durante l'operazione di lettura del file di configurazione, in quanto ci si riferisce alle tabelle per specificare le operazioni da svolgere sulle varie classi, attraverso delle stringhe di testo che contengono i nomi delle stesse.
  E' possibile costruire questa relazione utilizzando un attributo *Map <Value, Table>* che sfrutta una funzione di hashing per collegare il *Value* con l'istanza di *Table*: si può accedere ad una tabella attraverso il metodo *selectTableByName* se si ha a disposizione la stringa che specifica il nome della tabella; alternativamente, se si ha a disposizione il *Value*, si può sfruttare il metodo *selectTableByValue*.

- La classe **<RepositoryTables>** estende la classe *Tables*, definisce la collezione di tabelle che si riferiscono ad un dato repository e implementa metodi che potrebbero essere neces-

sari per la singola sorgente. Il nome di questa classe deriva anch'esso dal concatenamento fra il nome del repository e *Tables*: esso potrà essere, ad esempio, **EncodeTables** nel caso in cui la sorgente sia ENCODE. In questa classe è definito il metodo *getNewTables*, in cui vengono istanziate tutte le classi di tipo tabella che vengono immediatamente relazionate con il valore dell'Enumeration.

Nell'immagine 5.3 vengono presentate, tramite un digramma UML, le relazioni appena spiegate.

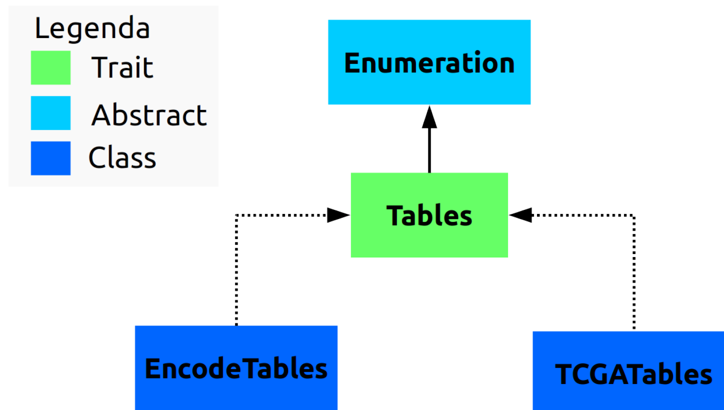


Figura 5.3: Diagramma UML Raccoglitore delle tabelle.

Capitolo 6

Funzionamento dell'applicativo

In questa sezione sarà presentata una panoramica riguardante il funzionamento del framework, sfruttando esempi pratici tratti dalle sorgenti **ENCODE** e **TCGA**.

6.1 Sorgenti supportate

Le sorgenti di dati per ora supportate dal framework sono due:

- **ENCODE**
- **TCGA**

Le caratteristiche e le singolarità delle due sorgenti sono presentate nella sezione 2.4. In seguito, se non esplicitamente specificato, le considerazioni descritte si riferiscono indistintamente ad entrambe le sorgenti.

6.2 Metodi di esecuzione

Le modalità di esecuzione del framework sono due:

- **Import** è la modalità con cui i dati, estrapolati dalle sorgenti, vengono inseriti nel database condiviso.
- **Export** è la modalità con cui i dati vengono esportati dal database condiviso e inseriti in file TSV che sono passati come input al framework, oppure la modalità con cui vengono creati nuovi file contenenti queste informazioni.

Di seguito, se non esplicitamente specificato, verrà considerata la modalità di esecuzione di **import**. Per chiarimenti sulla modalità di **export** si faccia riferimento alla Sezione 6.10.

6.3 Tecnica di Inserimento e Update di una tupla

Le chiavi univoche, descritte nella sezione 4.1, sono necessarie al momento dell'inserimento o dell'aggiornamento di un elemento all'interno della base di dati condivisa.
Dato un elemento da analizzare, il programma verifica, attraverso la chiave univoca, che esso non sia già presente all'interno del database: se l'elemento non risulta presente, viene effettuata un'operazione di inserimento e restituita la chiave primaria di tipo intero autoincrementale; viceversa, se l'elemento risulta già presente, viene effettuato un update sull'intera tupla, anche nel caso in cui gli attributi siano rimasti completamente immutati, e viene restituita la chiave primaria. Il valore di ritorno, rappresentante la chiave primaria della tupla inserita, potrebbe essere necessario come chiave esterna per attributi contenuti in altre entità.
L'efficienza funzionale di questo approccio è facilmente dimostrabile: infatti, il programma risulta essere perfettamente funzionante sia in caso il repository aggiorni l'identificativo univoco di una risorsa a fronte dell'aggiornamento di un attributo dei metadati, sia nel caso aggiorni solamente l'attributo senza modificarne l'identificativo. In quest'ultimo caso, viene semplicemente sovrascritto l'elemento già presente, mentre nel primo caso viene creato un nuovo elemento e il metodo di update restituisce la nuova chiave primaria della tupla, che può essere in seguito utilizzata per aggiornare i riferimenti delle chiavi esterne delle tuple nelle altre tabelle. Il procedimento di aggiornamento introduce, però, un inconveniente che risiede nella possibilità di lasciare delle tuple "orfane", cioè non collegate ad alcun altro elemento all'interno del database. Poiché a priori non si può determinare quale sia il metodo di aggiornamento utilizzato dalle varie sorgenti, si è scelto di implementare una soluzione di gestione degli inserimenti/update che sia più generica possibile.

6.4 Parametri in Input

Il programma prevede quattro parametri in input: il metodo di esecuzione del framework (*import* o *export*), il nome della sorgente a cui fanno riferimento i dati, il path ai file di dati (*data file*) e il path al file di configurazione (*configuration file*). In caso di modalità di *import*, i file, dopo essere stati elaborati dal framework, produrranno una sequenza di operazioni con lo scopo di determinare un certo numero di inserimenti o aggiornamenti di tuple sulla base dati condivisa.
Il path ai file di dati deve elencare una cartella con uno o più *data file* che possano essere elaborati dal framework: i *data file* sono selezionati per mezzo di un'*espressione regolare* (o *regex*) che ne specifica l'estensione e che dipende dalla sorgente di riferimento. Il framework prende in considerazione tutti i file che soddisfano la *regex*, selezionandoli dalla cartella indicata dal parametro in input e ricorsivamente in tutte le sue sottocartelle. È possibile osservare un esempio di parametri in input nel Listing 6.1. La figura 6.1 presenta graficamente ciò che si aspetta il framework come dato in ingresso, distinguendo le due modalità ed elencando ordinatamente gli input previsti.

```
1 import
2 encode
3 /home/federico/Scrivania/Encode_Download
```

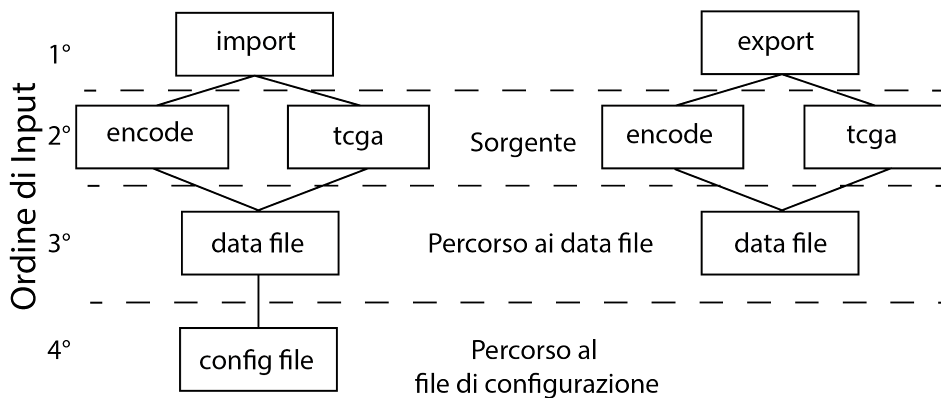


Figura 6.1: Schema dei parametri in input

```
4 /home/federico/GMQL/GMQL-Importer/Example/xml/setting.xml
```

Listing 6.1: Esempio di parametri in input

6.4.1 File dei dati

Il *data file*, ossia il file contenente i dati estrapolati dalla sorgente che si desidera importare sul repository condiviso, è funzionale alla creazione di associazioni fra gli attributi del repository sorgente e gli attributi descritti dal GCM.
Il formato del file non influisce sul funzionamento dell'applicativo: infatti, è possibile sviluppare routine di lettura dei file, indipendentemente dal formato.
Il formato di *data file* in input per ora supportato è un **tab-separated values (TSV)** con sole due colonne: la prima indica il valore chiave da inserire nel file di configurazione nel campo *sourceKey*, mentre la seconda indica il valore mappato dalla chiave che, dopo opportune trasformazioni, sarà inserito all'interno del database. Un estratto di *data file* in input è illustrato nel Listing 6.2.
Il framework predispone una collezione di tipo *Map[String, List[String]]* e la popola, attraverso il *data file*, leggendone ogni riga ed inserendo come chiave l'elemento della prima colonna e come valore l'elemento della seconda colonna. Al fine di permettere la mappatura di più *chiavi* identiche a molteplici valori, la collezione dispone di una lista di stringhe come valore mappato.
Se una riga non rispetta la struttura di tipo *chiave-valore*, utilizzando un carattere di *tab* come separatore, il framework la ignora, non inserendola nella collezione e loggando la riga non ben formata che è stata riscontrata. In caso di eccezioni impreviste in lettura del file, questo viene completamente ignorato: il suo nome viene loggato e una variabile contatore ne tiene traccia nel modulo delle statistiche (vedi Sezione 6.9 per maggiori informazioni sulle statistiche)
Una particolare considerazione risulta opportuna per il repository ENCODE: infatti, esso non riceve in input unicamente il file TSV ottenuto dall'esecuzione del *Downloader and Transformer* (descritto in Sezione 5.4), ma sfrutta anche le informazioni del file *json* originale, scaricato dalla sorgente, al fine di recuperare le informazioni sugli item (da cui il file *meta* deriva) che non

sono presenti nel file TSV. Questo comportamento è una singolarità del repository ENCODE, resa possibile dal fatto che per questa fonte dati sono consultabili la “storia” di un file elaborato e tutte le operazioni svolte per ottenerlo. Per maggiori dettagli sulla modalità di recupero di queste informazioni (mancanti) dal file json originale, si faccia riferimento alla Sezione 6.7.

```
file__content_md5sum 1625eff7b812daae706ce2613f56699e
file__@id /files/ENCFF112WBV/
replicates__1__experiment /experiments/ENCSR056UBA/
replicates__1__schema_version 8r
```

Listing 6.2: Un estratto di data file Encode

Struttura delle cartelle ed estensione dei *data file*

Il framework discrimina i file contenuti nella directory raggiunta attraverso il percorso specificato nel parametro in input (Linea 1 del Listing 6.1) e in tutte le sue sottocartelle, analizzando in modo ricorsivo l’estensione di ogni file e filtrando tramite espressione regolare.
Affinché i file possano essere presi in considerazione, l’estensione dei *data file* per il repository TCGA deve essere di tipo *.bed.meta*, mentre per la sorgente ENCODE deve essere di tipo *.bed.meta.json*: a sua volta, il file json di supporto deve essere di tipo *.bed.gz.json*. Se per TCGA la struttura delle cartelle non ha alcuna importanza, in quanto è sufficiente fornire in input la working directory per fare in modo che il framework iteri su tutti i file che soddisfano la *regex* che ne specifica l’estensione, ciò non si verifica per il repository Encode, in cui la struttura delle cartelle risulta essere essenziale; essa infatti, deve rispecchiare quella ottenuta dal *Downloader and Transformer* ed è composta da due directory allo stesso livello:

- **Downloads** è la cartella dove sono contenuti i json originali scaricati dalla sorgente.
- **Transformations** è la cartella in cui il *Downloader and Transformer* restituisce in output i file elaborati in formato TSV.

I nomi dei file, così come le estensioni, non devono essere modificati: è importante che ogni file trasformato abbia lo stesso nome, con estensione diversa, del file originale, in modo da consentire al framework di accedervi per recuperare le informazioni riguardanti gli item da cui il file *bed* deriva. Inoltre, ai fini del funzionamento della procedura di *export* dei dati, è essenziale che il nome del file sia uguale all’attributo *sourceId* della classe *Item* a cui fa riferimento, così da poter recuperare le informazioni dell’item interessato e scriverle nel file di destinazione corretto, nel caso in cui il framework sia utilizzato in modalità di *import*.
È buona norma dividere i *data file* in input al framework in più dataset, in modo che un’esecuzione non impieghi diverse ore per terminare e si possano ottenere le statistiche per verificarne il funzionamento. Ad esempio, il *Downloader and Transformer* fornisce in output i file suddivisi per *assembly* (hg19 e GRCh38) che a loro volta sono suddivisi, per formato dei file, in *broadPeak* o *narrowPeak*: al loro interno si trovano le sottocartelle Downloads e Transformations con i relativi file. Si può scegliere se eseguire il programma a livello di directory differenziate per *assembly* oppure a livello di directory differenziate per formato. Esecuzioni più ristrette (in

questo caso a livello di formato) permettono un'analisi più dettagliata delle statistiche di fine programma, per poter trarre considerazioni più accurate riguardanti dati di tipologia affine.

Stato dei file

Uno dei problemi affrontati nello sviluppo dell'applicazione è relativo all'inconsistenza di alcuni dati ottenuti dalla trasformazione dei metadati ENCODE *.json* in "file trasformati" *.bed.meta.json* utilizzando il tool *Downloader and Transformer*.
Alcuni di questi file, infatti, non disponevano delle chiavi univoche definite nel Capitolo 4: la mancanza di questi attributi ha comportato, nel momento dell'inserimento, uno stallo del software in quanto essi sono definiti come obbligatori all'interno dello schema concettuale. L'obbligatorietà è una proprietà fondamentale per questi attributi, considerando che tutte le operazioni di aggiornamento e inserimento dei dati si basano sui valori in essi contenuti. La non disponibilità di queste informazioni preclude la possibilità di distinguerli univocamente all'interno della base di dati.
Un primo tentativo di risolvere il problema è stato quello di escludere i file con status *archived*[1] o con stato non definito[2]. Nonostante questo filtro, il problema continuava a persistere: si è deciso, quindi, di effettuare un'analisi statistica sulla quantità dei file che non presentavano la struttura attesa: se la loro quantità fosse risultata esigua si sarebbe potuto continuare con l'esclusione completa degli inserimenti per tutte le tabelle relative a quello specifico file; se, invece, la quantità di dati fosse risultata molto elevata (mancanza della chiave primaria in diversi file per diverse tabelle), questo problema avrebbe comportato una perdita di dati troppo ingente e sarebbe stato necessario intraprendere una nuova strada. L'analisi ha dimostrato che il numero di file non contenenti queste informazioni era realmente esiguo, all'incirca l'1,2%, sulla totalità dei dati analizzati e minore dell'1,5% sulla totalità dei file *released*[3], come riassunto dalla Tabella 6.1.
Inoltre, un'ulteriore analisi ha permesso di verificare che la maggior parte di questi file non disponeva delle informazioni necessarie per il corretto inserimento all'interno del database già a livello della sorgente, probabilmente perché si tratta di esperimenti svolti al di fuori del progetto e da laboratori ENCODE, trasferiti solamente in un secondo momento. La tabella 6.1 riassume le statistiche collezionate durante esecuzioni del framework sui dati disponibili della sorgente ENCODE: si noti come a fronte di un'alta presenza di file di tipo *archived* o di stato non specificato vi è una bassissima presenza di file *realeased* ma non inseriti nel repository condiviso, a causa della mancanza degli attributi univoci a livello di data file in input.
Per quanto riguarda la sorgente TCGA non è stato riscontrato questo problema: in ogni esecuzione del programma si è osservato l'inserimento della totalità dei file a disposizione.

[1] Sono indicati come *archived* i file che sono considerati superflui o obsoleti dalla sorgente.

[2] Da un'analisi di questi file si è riscontrato che si tratta di file con stato non specificato, ma con struttura del tutto simile a quella dei file *archived*.

[3] Sono indicati come *released* i file che prevedono la completa disponibilità dei dati dell'esperimento a livello pubblico

Assembly	File analizzati	File released	File archived o stato non specificato	File released ma non iseriti	File inseriti
HG_19	12525	10840	1685	306	10534
GRCh38	12071	10091	1980	0	10091
Somma	24596	20931	3665	306	20625
		85,10%	**14,90%**	**1,24%**	**83,86%**

Tabella 6.1: Statistiche di riepilogo dello stato dei file per la sorgente ENCODE.

Data file con struttura non ben formata

Come accennato precedentemente, a causa di bug presenti nei tool sfruttati per la generazione dei data file in ingresso al framework, è possibile che vi siano dei file in ingresso che non rispettano la struttura specificata, cioè coppie di ⟨chiave, valore⟩ separate da un carattere di tab. Il metodo che implementa la lettura del file in ingresso prevede delle eccezioni non bloccanti durante la lettura del file, in modo che, anche in presenza di un errore di formattazione del data file in input, questo non comporti la perdita totale dell'inserimento delle informazioni. Per ora le eccezioni non bloccanti previste sono:

- righe che non prevedono un valore a fronte di un attributo, in questo caso la linea viene semplicemente ignorata dal framework passando alla riga successiva.
- righe che prevedono più valori a fronte di un attributo, anche in questo caso la riga viene ignorata passando al contesto del file successivo.

Altre tipologie di eccezioni comportano l'invalidazione del data file in input e il passaggio al contesto del file successivo. Entrambe le tipologie di eccezioni sono loggate a video, indicando il file e la motivazione del lancio dell'eccezione e memorizzate per motivi statistici, in modo tale che sia possibile indicare allo sviluppatore del tool eventuali errori di generazioni del file.
Durante l'esecuzione del framework si sono riscontrate unicamente malformità di formattazione in cui ad un attributo non veniva effettivamente rispecchiato un valore, in particolare sono state riscontrate malformità solamente nel caso di dati HG19, riepilogati nel Listing 6.3.

```
2018-03-11 00:12:47,653 INFO [main$] Total malformation found
   106
```

Listing 6.3: Esempio di statistiche di fine esecuzione

6.4.2 File di configurazione

Il file di configurazione ha lo scopo di mappare le informazioni contenute nel *data file* in ingresso verso la struttura del GCM attraverso la definizione di operazioni di mappatura. Uno degli scopi fondamentali del progetto consiste nello sviluppo di un framework di importazione dei dati che sia configurabile per quanto possibile dall'esterno, in maniera indipendente dall'implementazione. Per questo è stato progettato un file di configurazione che possa rendere indipendenti dal codice scritto i futuri cambiamenti sulle scelte di mapping degli attributi all'interno dello schema.
Il formato del file di configurazione, per ora utilizzato per le sorgenti **Encode** e **TCGA** è di tipo

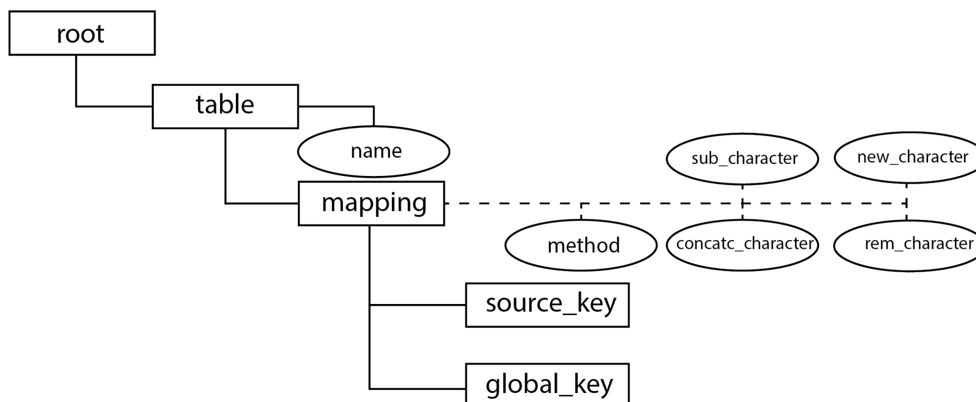


Figura 6.2: Rappresentazione grafica della struttura del file di configurazione.

XML.
Il file è composto da 3 livelli:

- Table
  - Mapping
    - SourceKey
    - GlobalKey

La struttura del file XML è definita nell'Appendice C. Una routine di verifica della correttezza del file di configurazione viene eseguita non appena il file viene letto.
Il livello **Table** contiene tutti i mapping che si riferiscono ad una tabella, utilizzando una proprietà di tipo *name* obbligatoria.
Il livello **Mapping** descrive una singola mappatura fra un campo del file sorgente e un attributo della tabella di riferimento. Questo tag può prevedere diversi attributi, la cui definizione implica diversi comportamenti della mappatura a livello di codice. I possibili attributi sono, ad esempio:

- METHOD è l'attributo principale del tag e descrive la tipologia di inserimento del dato all'interno del repository condiviso. Uno tra i possibili metodi di inserimento è "CONCAT" (concatenamento), ad esempio nel caso in cui siano previsti più mapping riferiti alla stessa *SourceKey*: questo metodo è stato impiegato anche per implementare la proprietà multivalore descritta nel Capitolo 3. È possibile anche descrivere manualmente il valore di un attributo, utilizzando il metodo "MANUALLY" e inserendo nel campo *SourceKey* il valore desiderato: questo metodo è utile nel caso in cui il repository sorgente non preveda un attributo presente sul GCM e sia quindi necessario definire un valore di default per tutte le tuple di quel repository. Nel caso in cui l'attributo METHOD non venga definito, subentra un metodo "DEFAULT" che semplicemente mappa il contenuto della sorgente nell'attributo indicato, andando a sovrascrivere eventuali valori già presenti. Nella tabella 6.2 sono elencanti i metodi di inserimento sviluppati nella versione attuale.
  Nell'eventuale bisogno di un nuovo metodo di inserimento, è possibile definire quest'ultimo

in qualsiasi momento, andando a modificare la classe **InsertMethod**, di tipo Singleton, e specificandolo opportunamente nel file di configurazione (per una descrizione più dettagliata della procedura fare riferimento al Capitolo 7).

I seguenti attributi assumono funzionalità solamente nel caso in cui l'attributo METHOD sia stato definito: in caso contrario, la loro definizione non influenza in alcun modo il comportamento dell'inserimento in modalità di "DEFAULT". Inoltre, la loro definizione assume significato solamente con alcuni metodi (vedi Tabella 6.3): in tutti gli altri casi, anche se definiti, non apportano alcuna alterazione al comportamento del processo di inserimento.

- CONCAT_CHARACTER permette di definire la stringa di concatenamento per tutti i metodi di tipo "CONCAT"; se non definito viene utilizzato il carattere spaziatore.

- SUB_CHARACTER permette di definire la stringa da sostituire in tutti i metodi di tipo "SUB". Se in un metodo di tipo "SUB" questo attributo non viene definito non avviene alcuna sostituzione.

- NEW_CHARACTER permette di definire la stringa che verrà sostituita a SUB_CHARACTER in tutti i metodi di tipo "SUB"; se non definito, viene utilizzato il carattere nullo. Si noti quindi che a fronte della definizione del SUB_CHARACTER e una non definizione di NEW_CHARACTER, il comportamento ottenuto è quello tipico di una cancellazione.

- REM_CHARACTER permette di definire la stringa da eliminare in tutti i metodi di tipo "REMOVE".

Gli attributi SUB_CHARACTER, NEW_CHARACTER e REM_CHARACTER possono definire una lista di stringhe, divise da un carattere speciale (di default il carattere '*'), in modo da effettuare molteplici operazioni di rimozione o sostituzione per un singolo mapping. Ad esempio REM_CHARACTER="/*!*labs" rimuove tutte le occorrenze dei caratteri '/' e '!' e delle stringhe "labs". Ovviamente il numero di stringhe o caratteri esplicitati negli attributi NEW_CHARACTER e SUB_CHARACTER deve essere il medesimo per ottenere il corretto funzionamento della funzione di inserimento.
Gli elementi della coppia **SourceKey** e **GlobalKey** rappresentano rispettivamente la chiave del valore desiderato nel file di input e l'attributo della tabella di riferimento in cui il valore deve essere mappato.
L'immagine 6.2 presenta graficamente la struttura del file di configurazione appena presentato. È possibile definire all'interno dell'attributo METHOD una composizione dei metodi illustrati nella tabella 6.3, in modo da ottenere più effetti per una singola mappatura. I metodi devono essere divisi da un carattere di trattino ('-') ed essi sono eseguiti sequenzialmente partendo dal metodo più a sinistra per terminare con quello più a destra. Ad esempio, il metodo definito come "REMOVE-SUB-CONCAT" prima rimuove le stringhe indicate nei parametri REM_CHARACTER, dopodiché sostituisce i termini SUB_CHARACTER con NEW_CHARACTER ed infine concatena la stringa ottenuta con l'attuale utilizzando come carattere divisore CONCAT_CHARACTER. È possibile visionare una componente parziale del file di configurazione nel

Listing 6.4.

Si faccia attenzione a non confondere le operazioni di mappatura indicate nel file di configurazione con le regole di Contestualità o Dipendenza spiegate nella Sezione 4.4: le seconde, infatti, sono regole valide per tutte le sorgenti di dati e parte integrante del modello concettuale del sistema, mentre le prime sono scelte dall'utente e funzionali alla sorgente di riferimento.

Tabella 6.2: Metodi disponibili di inserimento degli attributi

Metodo	**Descrizione**
Default	Assegna il nuovo parametro sostituendo quello vecchio se presente
Manually	Assegna al parametro il valore definito dall'utente
Concat	Concatena il nuovo parametro a quello attuale utilizzando il carattere indicato
CheckPrec	Assegna il nuovo parametro solo se il parametro attuale è uguale al valore nullo
Remove	Rimuove la stringa indicata
Sub	Sostituisce la stringa indicata con una nuova stringa
Uppercase	Converte la stringa in lettere maiuscole
Lowercase	Converte la stringa in lettere minuscole

Tabella 6.3: Esempi di metodi disponibili per l'inserimento degli attributi: il simbolo ✓ indica che per il metodo ha senso definire l'attributo, mentre la × indica che se anche l'attributo venisse indicato questo non produrrebbe nessun effetto alla procedura di mapping

Metodo	CONCAT_CHARACTER	SUB_CHARACTER	NEW_CHARACTER	REM_CHARACTER
Default	×	×	×	×
Manually	×	×	×	×
Concat	✓	×	×	×
Manually-Concat	✓	×	×	×
CheckPrec	×	×	×	×
Sub-Concat	✓	✓	✓	×
Remove-Concat	✓	×	×	✓
Remove-Sub-Concat	✓	✓	✓	✓

```
1 <root>
2  <table name="CONTAINERS">
3          <mapping method="SUB-CONCAT" sub_character=" "
              new_character="_" concat_character="_">
4             <source_key>file__assembly</source_key>
5             <global_key>name</global_key>
6          </mapping>
7          <mapping method="SUB-CONCAT" sub_character=" "
              new_character="_" concat_character="_">
8             <source_key>award__project</source_key>
```

```
          <global_key>name</global_key>
       </mapping>
       <mapping method="SUB-REMOVE-CONCAT" rem_character="
          Peak" sub_character=" " new_character="_"
          concat_character="_">
          <source_key>file__file_format_type</source_key>
          <global_key>name</global_key>
       </mapping>
       <mapping>
          <source_key>file__assembly</source_key>
          <global_key>assembly</global_key>
       </mapping>
       <mapping method="MANUALLY">
          <source_key>false</source_key>
          <global_key>isAnn</global_key>
       </mapping>
       <mapping method="MANUALLY">
          <source_key>null</source_key>
          <global_key>annotation</global_key>
       </mapping>
 </table>
 <table name="ITEMS">
    <mapping>
       <source_key>file__accession</source_key>
       <global_key>sourceId</global_key>
    </mapping>
    <mapping>
       <source_key>file__output_type</source_key>
       <global_key>dataType</global_key>
    </mapping>
    <mapping>
       <source_key>file__file_type</source_key>
       <global_key>format</global_key>
    </mapping>
    <mapping>
       <source_key>file__file_size</source_key>
       <global_key>size</global_key>
    </mapping>
    <mapping>
       <source_key>file__href</source_key>
       <global_key>sourceUrl</global_key>
    </mapping>
  </table>
```

```
50 </root>
```

Listing 6.4: Esempio di parte del file di configurazione per il repository ENCODE per le entità Container e Item

6.4.3 XMLReader

Il compito della classe **XMLReader** è quello di elaborare le informazioni provenienti dal file di configurazione XML e fornire in output una lista di operazioni che corrispondono all'unione tra le specifiche di mapping e i metodi di inserimento definiti nella tabella 6.2: il numero di operazioni contenute nella lista sarà direttamente proporzionale al numero di coppie *SourceKey-GlobalKey* presenti all'interno del file XML.

6.4.4 Sequenza di operazioni

Ogni operazione non corrisponde strettamente ad un inserimento o aggiornamento di una tupla all'interno del database, in quanto diverse operazioni possono riferirsi allo stesso attributo, sfruttando per esempio il metodo di concatenamento. Ogni operazione è descritta dalla quadrupla: <NOME_TABELLA, CHIAVE_DELLA_SORGENTE, ATTRIBUTO_TABELLA, METODO_DI_INSERIMENTO>.
Un esempio di sequenza di operazioni è illustrato in Figura 6.3.
Una volta completata la procedura della creazione delle operazioni, queste vengono lette in sequenza dal **Main** del programma che mappa la CHIAVE_DELLA_SORGENTE, sfruttando la collezione *Map[String, List[String]]*, al suo valore di target. Il valore mappato, insieme al nome della tabella, all'attributo e al metodo di inserimento, è passato al *Raccoglitore di tabelle* (vedi Sezione 5.5.3) che si occupa di popolare, sfruttando il valore target, l'attributo della tabella attraverso il metodo di inserimento indicato.
Una volta completata la routine di popolazione degli attributi delle tabelle del modello, si passa alla fase di inserimento/aggiornamento delle tuple.
L'ordine della sequenza di operazioni, che rispecchia l'ordine di scrittura delle regole di mappatura nel file di configurazione, non è importante ai fini dell'inserimento degli elementi del database: infatti, un ordine prestabilito e costante di inserimento delle tabelle è già descritto nel modulo Raccoglitore di Tabelle (vedi sezione 5.5.3) e, poiché questo ordine garantisce il corretto ottenimento delle chiavi esterne, è necessario che venga rispettato. Quindi, indipendentemente da come viene scritto il file di configurazione, sarà il framework a preoccuparsi di selezionare le tabelle secondo la sequenza fissa prestabilita dal gestore e sfruttata per l'inserimento delle tuple.

L'esistenza di un ordine di inserimento è necessaria poiché esistono dipendenze tra una tabella e l'altra: infatti, nel momento in cui una tabella viene inserita, il programma restituisce un intero che è la sua chiave primaria ma che potrebbe essere un prerequisito necessario per l'inserimento di una seconda tabella, in quanto potrebbe essere la sua chiave esterna.
L'ordine di inserimento delle tabelle è il seguente: Donor, BioSample, Replicate, ExperimentType, Project, Container, Case, Item, DerivedFrom, CaseItem e ReplicatesItem. L'ordine di inserimento non è unico: infatti, è possibile definire un qualsiasi ordine di inserimento che rispetti

le dipendenze che intercorrono fra le varie tabelle, descritte nella Figura 6.4. Per rispettare le dipendenze è necessario che, data una tabella che si vuole inserire all'interno del database, tutte le tabelle collegate ad essa con arco uscente siano state già inserite all'interno del database.

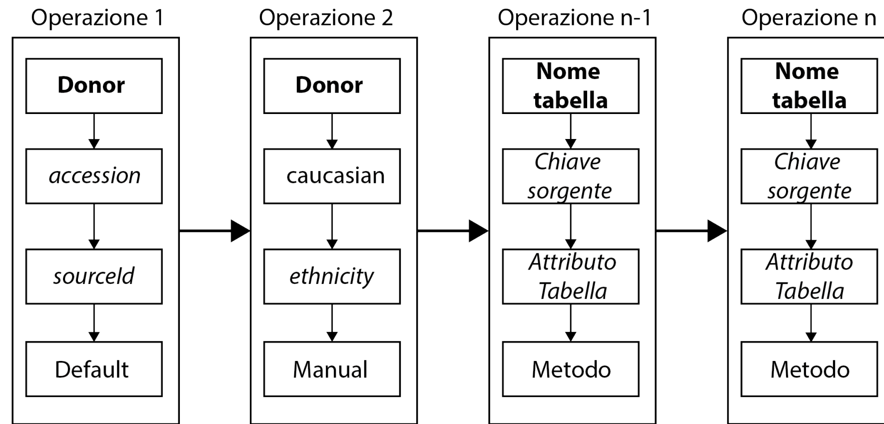


Figura 6.3: Rappresentazioni di possibili operazioni

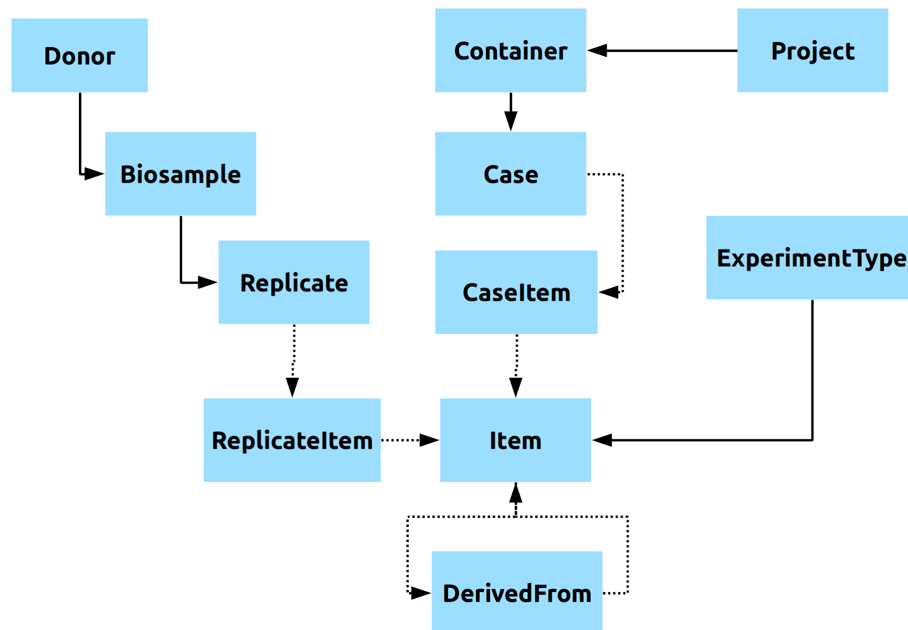


Figura 6.4: Dipendenze delle tabelle nella procedura di export

6.5 Schema logico globale del framework

Riassumendo, come prima operazione la classe **Main** verifica la correttezza del file XML in ingresso, attraverso un file *xsd* caricato sul repository GitHub del progetto; dopodiché essa istanzia una classe **XMLReader** a cui darà in input il file di configurazione. Una volta ottenuta in output la lista delle operazioni la classe Main crea la collezione di mapping di tutte le righe del file dei dati in input del tipo *Chiave* $\rightarrow$ *[Valore]* (sfruttando la collezione *Map[String, List[String]]* specificata prima), così da ottenere il valore di una chiave del file in input con costo medio di ricerca indipendente dal numero di elementi. È essenziale che un singolo valore di chiave sia associato ad una lista di stringhe, in modo che chiavi omonime nel file in input siano tutte prese in considerazione.
La classe Main istanzia una classe di tipo **<RepositoryTables>** in base al repository di riferimento e richiama, per ogni elemento contenuto nella lista delle operazioni, il metodo *populateTable* che ha il compito di "popolare" l'attributo <ATTRIBUTO_TABELLA> della tabella <NOME_TABELLA> attraverso il mapping relativo alla <CHIAVE_DELLA_SORGENTE> utilizzando il <METODO_DI_INSERIMENTO> specificato.
Terminata la lista di operazioni la classe Main richiama il metodo *insertTables* della classe **<RepositoryTables>**: con questa azione ha inizio la routine di inserimento o aggiornamento dei valori appena inseriti all'interno delle istanze delle classi delle tabelle. Completate tutte le operazioni di inserimento o aggiornamento, vengono mostrate a video le statistiche collezionate durante l'esecuzione del framework.
Il software esegue in sequenza le tre fasi del processo ETL per ogni singolo file, prima di passare al contesto del file successivo. Il funzionamento di alto livello del framework è descritto dallo pseudo-codice contenuto nell'Algoritmo 1.

6.6 Implementazione delle relazioni e dipendenze fra tabelle

Il modello concettuale presentato nel Capitolo 4 definisce le entità e le relazioni fra esse. Nel framework queste relazioni sono esplicitate in due locazioni: nel **DBHandler** e nelle classi ***<NomeTabella>***. Nel **DBHandler** le relazioni sono esplicitate tramite la definizione di indici che vanno ad elencare le relazioni di chiavi esterne fra una tabella e l'altra e che sono poi utilizzati al momento della creazione della struttura delle tabelle del database andando quindi a modificare il DDL di ogni singola tabella. In questo caso è esplicitata la relazione "fisica" delle tabelle, attribuendo agli attributi le proprietà specifiche di un database relazionale. Le relazioni "logiche" all'interno del framework sono invece esplicitate da attributi presenti nella classe *<Table>* (di tipo Trait) e modificate, in caso di necessità, dalle classi Trait che specificano le caratteristiche di ogni singola tabella: in particolare, è presente un attributo booleano chiamato *_hasForeignKeys* che specifica se ogni singola tabella ha nelle sue proprietà delle chiavi esterne, e che di default è settato a `false`. Quando questo attributo è settato a valore `true`, un altro attributo, di nome *_foreignKeysTables*, definisce, tramite una lista di stringhe, i nomi delle tabelle con le quali la tabella ha una relazione; l'attributo, essendo una lista di stringhe (che di default è settata a valore nullo), può definire un numero illimitato di relazioni. Nella procedura di inserimento delle tuple il Gestore delle Tabelle controllerà se la tabella corrente ha

Algorithm 1: Inserimento o Aggiornamento tuple nel Database.

```
Data: [DataFilePath, ConfigFile]
Result: Inserimento o aggiornamento tupla nella modalità di import
Inizializzazione;
if Il ConfigFile è corretto then
    while Per ogni file contenuto nel DataFilePath che rispetta la regex do
        while Per ogni mappatura contenuta nel ConfigFile do
            Crea la nuova operazione;
            Aggiungi in coda l'operazione alla catena delle operazioni;
        Crea la struttura Map[String, List[String]] usando il data file;
        while Per tutte le operazioni contenute nella catena do
            Popolare l'attributo <ATTRIBUTO_TABELLA> della classe
             <NOME_TABELLA> attraverso il mapping relativo alla
             <CHIAVE_DELLA_SORGENTE> utilizzando il
             <METODO_DI_INSERIMENTO>;
        Ottenere l'ordine di inserimento delle tabelle;
        while Le tabelle non sono finite do
            if Se la chiave univoca non è presente nel DB then
                Inserire la tupla nel database e ritornare la chiave primaria;
            else
                aggiornare la tupla nel database e ritornare la chiave primaria;
else
    Esci con un errore;
```

qualche relazione tramite il valore booleano *_hasForeignKeys*: se questo parametro risulta falso si passa direttamente all'inserimento della tupla, mentre se questo parametro risulta vero il Gestore delle Tabelle richiama, per ogni tabella definita nella lista *_foreignKeysTables*, il metodo *setForeignKeys(table)*, passando come parametro il nome della tabella definita nella lista. A questo punto, visto che ogni tabella salva la sua chiave primaria in un attributo definito in <*Table*> chiamato *PrimaryKey*, viene assegnato al parametro di chiave esterna il valore di chiave primaria della tabella da cui essa dipende. Si può intuire da questa spiegazione che la definizione del modello delle tabelle del codice è del tutto indipendente dalla struttura fisica del modello: infatti, è sufficiente ridefinire i legami all'interno della lista in modo tale che il codice non si accorga del cambio del modello concettuale. Ovviamente, per rendere il codice più chiaro e comprensibile, a fronte del cambio di una relazione è consigliato il cambio dei nomi degli attributi che definisco le chiavi esterne. Si può osservare un esempio nel Listing 6.5, mentre l'algoritmo 2 illustra in pseudocodice la procedura appena indicata.
Un approccio del tutto simile è stato sfruttato per implementare il controllo *debole* delle dipendenze fra le varie tabelle. Il valore booleano *_hasDependencies* indica se una determinata tabella ha dipendenze e, nel caso le avesse, queste sono elencate nella lista *_dependenciesTables*. Si noti

che in questo caso una tabella può dipendere anche da se stessa, ad esempio per Dipendenze di esistenza obbligatoria con entrambi gli attributi della stessa tabella (Biosample.*type* = "tissue" $\longrightarrow$ Not Null(*tissue*)). Un esempio è illustrato dal Listing 6.6.
Entrambe queste definizioni, essendo contenute nella classe Trait ***<NomeTabella>***, non necessitano di ridefinizione nel caso si volesse aggiungere il supporto ad una nuova sorgente di dati, ma esse sono semplicemente ereditate dalla nuova classe definita.

Algorithm 2: Controllo dipendenze delle chiavi esterne

Popola tutte le istanze con i dati dei *data file* specificati nel *configuration file*;
Ottenere l'ordine di inserimento delle tabelle;
while *Fino a che le tabelle non sono finite* **do**
  if *Se la tabella ha dipendenze (hasDependecies)* **then**
    while *Per ognuna delle tabelle contenuta nella lista delle dipendenze* **do**
      Chiama il metodo *setForeignKeys*;
  Richiama il metodo *InsertRow* implementato dalla classe;

```
_hasForeignKeys = true

_foreignKeysTables = List("DONORS")

override def setForeignKeys(table: Table): Unit = {
   this.donorId = table.primaryKey
}
```

Listing 6.5: Estratto di codice della classe Biosample che specifica le relazioni di chiavi esterne e relativi metodi

```
_hasDependencies = true

_dependenciesTables = List("CONTAINERS", "DONORS")

override def checkDependenciesSatisfaction(table: Table): Boolean = {
   try {
     table match {
       case container: Container => {
         if (container.isAnn && container.annotation == null) {
           Statistics.constraintsViolated += 1
           this.logger.warn("Container annotation constrains violated")
           return false
         }
         true
       }
       case donor: Donor => {
         if (donor.species.toUpperCase().equals("HOMO SAPIENS") &&
             !(this.assembly.equals("hg19") ||
             this.assembly.equals("GRCh38"))) {
```

```
      Statistics.constraintsViolated += 1
      this.logger.warn("Container species constrains violated")
      return false
     }
     true
    }
    case _ => true
   }
  } catch {
   case e: Exception => {
    logger.warn("java.lang.NullPointerException")
    true
   };
  }
}
```

Listing 6.6: Estratto di codice della classe Container che specifica le dipendenze della tabella

6.7 Molteplici Item, Replicati e Donatori per ogni file in Encode

Per il repository ENCODE, come precedentemente accennato, ad ogni singolo file *bed* possono corrispondere molteplici file *fastq* o *bam*: tenendo conto dell'intenzione di inserire anche le informazioni di file non processati, vi è quindi la possibilità di inserire molteplici tuple nella tabella Item per ogni file di partenza. Inoltre, considerando che ogni Item può riferirsi a diversi replicati, sia biologici che tecnici, e che questi replicati possono provenire da diversi donatori nel caso in cui l'esperimento sia di tipo *anisogenic*[4], si è reso indispensabile prevedere la possibilità di inserire più replicati, donatori e biosample per ogni *data file* in input. Se per quanto riguarda donatori, campioni biologici e replicati la disponibilità di informazioni era evidente a livello di file in input (è stata necessaria comunque l'implementazione di una logica che permettesse di "collegare" correttamente ogni donatore al proprio biosample e ogni Biosample al corretto replicato) questo non era vero, invece, per le informazioni degli antenati dell'item a cui il file faceva riferimento, poiché in questo caso i dati non erano disponibili a livello di *bed.meta.json* ma nel file originale. La tabella 6.4 rappresenta, tramite una media, il numero di item che sono inseriti o aggiornati per ogni singolo file di ENCODE: si può notare come mediamente ad ogni file corrispondono all'incirca sette operazioni di inserimento o aggiornamento di item nel repository condiviso, azioni che determinano, come spiegato nella Sezione 6.14, un dispendio più ingente di risorse.

6.7.1 Ricerca degli attributi Platform e Pipeline

Tra i vari attributi presentati nelle precedenti sezioni, si vuole porre una particolare attenzione agli attributi *Pipeline* e *Platform*.

---

[4]Si tratta di esperimenti in cui vengono prodotti diversi replicati provenienti da biosample di tessuto simile ma diverso donatore (per dettagli, `https://www.encodeproject.org/data-standards/terms/replication`)

Assembly	File Inserted	Item Inserted	Item Updated	Total Item	Item/File
HG_19	10534	16884	56370	73254	6,95
GRCh38	10091	24049	49533	73582	7,29
	20625	40933	105903	146836	7,12

Tabella 6.4: Statistiche di riepilogo di numero di item inseriti o aggiornati per il repository ENCODE.

Si ricorda che il valore *Pipeline* indica i vari processi che, nel corso dell'esperimento, i file di dati non processati derivati da campioni biologici hanno subito per essere trasformati in dati finali, mentre *Platform* indica la tipologia di macchina utilizzata per sequenziare il campione. Entrambi i valori sono gli unici, fra tutti gli attributi contenuti nelle entità, a non essere presenti nel file *bed.meta.json* fornito in output dal software *Downloader and Transformer*: la loro mancanza è giustificata dal fatto che solitamente l'attributo *Platform* è peculiare dei file non processati di tipo *fastq* (in quanto si tratta di file di tipo *raw*, primi nella catena della generazione dell'esperimento), mentre l'attributo *Pipeline* è tipico dei file semi-processati di tipo *bam*; nonostante ciò, esistono casi eccezionali in cui l'attributo *Platform* o l'attributo *Pipeline* risulta presente anche per file di tipo *bed*. In queste situazioni eccezionali, nel caso in cui risulta presente l'attributo *Pipeline* significa che il file analizzato non è l'ultimo nella catena dell'esperimento ma è un file intermedio; al contrario, se risulta presente l'attributo *Platform*, significa che il file è il primo presente nella catena dell'esperimento: il verificarsi di quest'ultima situazione, anche se possibile, è molto raro. L'assenza delle informazioni di *platform* e *pipeline* nel file dato in output dal *Downloader and Transformer* ha determinato la necessità di sviluppare due processi specifici per estrapolare questi dati, sfruttando il file *json* originale come supporto. Le routine di ricerca consistono, dopo aver recuperato eventuali valori di *Pipeline* o *Platform* dell'item primario, cioè quello a cui fa direttamente riferimento il *data file* in input, nell'analizzare il file *json* originale <ID>*.bed.gz.json* invece del file <ID>*.bed.meta.json* trasformato: infatti, il file *json* contiene un elemento, non presente nel file trasformato, identificato dalla chiave `"derived_from"`, che permette di identificare le dipendenze di un dato file *bed* e di individuare gli item non processati da cui esso deriva, tenendo traccia dei legami attraverso la relazione di autoanello creata sull'entità **Item**. La routine si compone dei seguenti passaggi:

1. Una volta inserite nella classe le informazioni dell'item **bed**, che fanno riferimento al file *sourceId*.bed.meta.json contenuto nella cartella Transformations utilizzando il processo precedentemente spiegato, restituire alla classe Item chiamante i possibili valori *Pipeline* o *Platform* del file *bed* considerato, passare quindi al file *sourceId*.bed.gz.json nella cartella Downloads. Notare l'importanza del nome per poter accedere al file corretto.

2. Individuare l'elemento `"@id"`: "/file/$sourceId$" e verificare la presenza dell'elemento `"derived_from"`:

    - nel caso esista, ricavare la lista dei file del tipo [$sourceId_1$, $sourceId_2$, ... , $sourceId_N$].

- nel caso non esista, uscire dalla funzione.

3. Per ogni file della lista andare nell'elemento del file json `"@id"`: "/file/$sourceId_i$" e ottenere tutti i dati necessari per il suo inserimento nel database.

    - Se il file è di tipo *fastq* o *bed*, ottenere anche l'attributo *Platform*.

    - Se il file è di tipo *bam*, ottenere anche l'attributo *Pipeline*.

4. Inserire o aggiornare l'Item nel database e tenere traccia della relazione con l'Item figlio nella tabella Derived From, inserendo come *initialId* la propria chiave primaria e come *finalId* la chiave primaria del file derivato; inoltre, inserire nella colonna *operation* il tipo di operazione tecnica eseguita per la processazione dell'item.

5. Ripartire dal passaggio 2 passando come parametro: $sourceId_i$.

La ricerca dei file derivati è modellata come una **Ricerca in Profondità** (*depth-first search*) di un *Direct acyclic Graph* (DAG). Un esempio in pseudo codice del procedimento è illustrato negli algoritmi 3 e 4. Si può notare come nell'algoritmo 3 vi sia l'impostazione del procedimento, con il recupero dei possibili valori *Pipeline* e *Platform* dell'item di formato *bed* e della lista dei suoi antenati, seguita poi dell'inizio della procedura ricorsiva (Algoritmo 4) che itera lo stesso processo sugli antenati del file *bed* e di conseguenza sui possibili file da cui essi derivano.

Algorithm 3: Procedura di recupero dei Derived Items

Data: [BEDITEM]
Result: Inserimento o aggiornamento dei Derived Items
Recuperare eventuale valore di *Pipeline* o *Platform*;
Recuperare il *sourceId* dal BEDITEM;
Aprire il file *sourceId*.bed.gz.json nella cartella Downloads;
Andare alla sezione “@id”: “/file/*sourceId*”;
if *se esiste* `"derived_from"` **then**
    Ottenere la lista dei file "padre" $sourceId_1, sourceId_2, \ldots, sourceId_N$;
    while *La lista non è terminata* **do**
        if *Item non è stato già visitato* **then**
            Richiamare: Recupero ricorsivo Derived Items($sourceId_i$);
else
    Esci;

Algorithm 4: Recupero ricorsivo Derived Items

Data: [*sourceId*]
Result: inserimento o aggiornamento dei Derived Items
Andare alla sezione “@id”: “/file/*sourceId*”;
Recuperare tutte le informazioni per inserimento dell’Item;
if *dataType è fastq o bed* **then**
    Recuperare anche *Platform*;
if *dataType è bam* **then**
    Recuperare anche *Pipeline*;
Inserire o aggiornare l’Item recuperato;
Tenere traccia nella tabella DerivedFrom della relazione fra item padre ed item figlio, indicato anche il tipo di operazione;
if *se esiste* `"derived_from"` **then**
    Ottenere la lista dei file "padre" $sourceId_1, sourceId_2, \ldots, sourceId_N$;
    while *La lista non è finita* **do**
        if *Item non è stato già visitato* **then**
            Richiama: Recupero ricorsivo Derived Items($sourceId_i$);
else
    return;

6.8 Replicati biologici e Replicati tecnici

Molte sorgenti di dati non distinguono il concetto di biosample, cioè replicato biologico, dal concetto di replicato tecnico: infatti, questa differenza è stata riscontrata ad ora solamente in

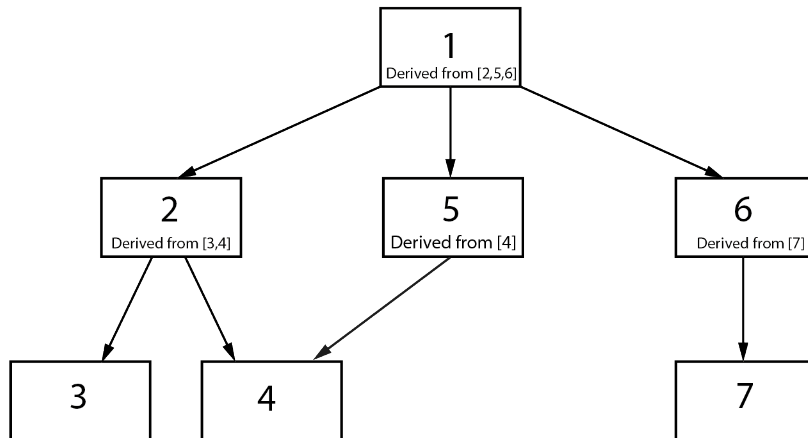


Figura 6.5: Rappresentazione del grafo aciclico orientato.

ENCODE. La sottile distinzione tra replicato biologico e tecnico è facilmente intuibile: per replicato biologico si intende un campione biologico estratto dal donatore, che successivamente può essere sezionato, a seconda dell'analisi che deve essere svolta, in uno o più replicati tecnici, rappresentati da sezioni del tessuto prelevato, le quali hanno subito modificazioni chimiche o fisiche funzionali allo svolgimento di analisi specifiche. Considerando la possibilità che nel contesto di un esperimento della sorgente ENCODE possano essere usati diversi replicati, sia biologici sia tecnici, provenienti potenzialmente da diversi donatori, vi è la necessità di trovare un modo per distinguere i vari replicati: per questo, ENCODE numera i replicati biologici con un valore intero crescente e concatenato, per mezzo di un carattere di underscore, a un altro valore intero che identifica il replicato tecnico di quel campione. Ipotizziamo un esperimento che preveda il prelievo di due campioni biologici da un donatore: a entrambi i campioni biologici è associata una chiave unica (*uuid*), utile a identificare il campione biologico nell'intera base di dati, e un numero intero (tipicamente progressivo), utile a identificare il campione nell'esperimento e alla distinzione dei due campioni biologici, che chiameremo B-1 e B-2. Nell'esperimento, il campione B-1 è suddiviso, per esigenze di analisi, in 3 campioni tecnici, nominati T-1_1 T-1_2 T-1_3, dove il primo numero indica il numero intero associato al campione biologico dell'esperimento, mentre il secondo numero serve per discriminare diversi replicati tecnici associati allo stesso campione; inoltre, ad ogni replicato tecnico è associata una chiave univoca (uuid) utile per distinguerlo all'interno dell'intera base di dati. Il secondo campione B2, ipotizzando che non sia stato ulteriormente suddiviso, avrà semplicemente un unico replicato tecnico chiamato T-2_1. Queste relazioni sono rappresentate dall'immagine 6.6. In ENCODE, come è possibile visionare da Appendice A, la chiave univoca del Replicate coincide con lo *uuid* del replicato a livello tecnico, mentre la chiave univoca del Biosample coincide con il valore dello *uuid* del replicato biologico. Nelle sorgenti che non prevedono la differenziazione del concetto di replicato tecnico e biologico, come TCGA, si è scelto di inserire, come chiave univoca per l'entità Replicate, la stessa utilizzata per la chiave univoca dell'entità Biosample, inserendo manualmente il valore '1' sia per il numero intero del replicato biologico sia per quello del replicato tecnico (vedere

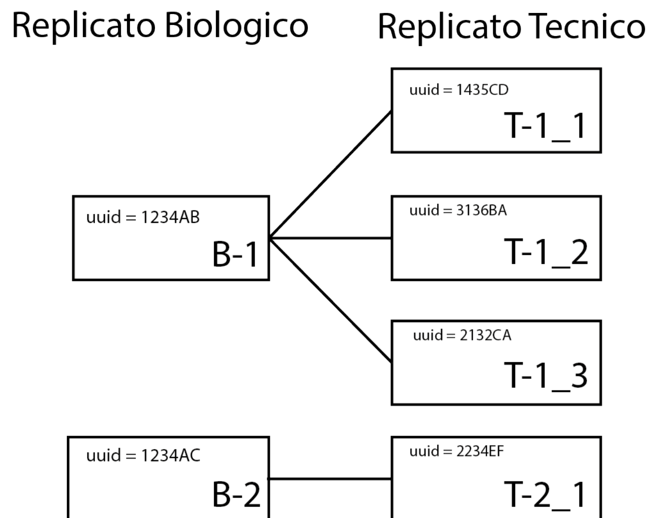


Figura 6.6: Esempio di relazione fra replicati biologici e tecnici.

Appendice B). In questo modo si è creata una tabella ponte che non porta alcuna informazione, ma che al contempo non sfrutta strani artifici per la definizione di attributi non presenti a livello di sorgente.

6.9 Statistiche e messaggi loggati

Durante l'esecuzione del framework, è possibile osservarne l'andamento per mezzo di messaggi di *log* stampati a video e contestualmente salvati in un file di log con nome univoco. I messaggi permettono di verificare il corretto avvenimento di un'esecuzione e informano l'utente riguardo a potenziali errori sia a "runtime" che in fase di settaggio dei parametri in input: l'utente, anche se inconsapevole delle modalità di esecuzione, è guidato nelle impostazioni iniziali. Di utilità significativa risultano anche le statistiche riassuntive di fine esecuzione, in quanto permettono di trarre considerazioni generali sull'andamento riassunte in poche righe.
I messaggi di log sono suddivisi in tre categorie:

- **INFO**: sono messaggi che informano l'utente riguardo la normale esecuzione del programma, in modo da favorire la comprensione del processo in atto o riguardo i parametri necessari per eseguire correttamente determinate operazioni.

- **WARN**: sono messaggi che informano l'utente riguardo situazioni anomale ma previste, che sono quindi gestite e mostrate a video. A seguito di questi messaggi non è previsto uno stallo del programma.

- **ERROR**: sono messaggi che informano l'utente di situazioni anomale o impreviste, quali

errori nell'input dei dati. Solitamente questi errori sono seguiti da uno stop controllato del programma.

Seguendo l'esecuzione logica del programma, i primi messaggi di errore potrebbero comparire quando si sbagliano a inserire i parametri in input (vedi Listing 6.7, 6.8 e 6.9): in questo caso, alcuni messaggi di **INFO** suggeriscono all'utente le possibili soluzioni al problema.

```
2018-02-12 21:39:38,844 [ERROR|main$] Incorrect number of arguments
2018-02-12 21:39:38,844 [INFO|main$] GMQLImporter help: Run with configuration_xml_path, file_folder, repository_ref and execution_mode as arguments
```

Listing 6.7: Esempio di errore nel numero di parametri passati in input

```
2018-02-12 22:11:51,678 [ERROR|main$] Incorrect execution_mode argument
2018-02-12 22:11:51,678 [INFO|main$] Please select 'import' or 'export'
```

Listing 6.8: Esempio di errore per il parametro 'execution'

```
2018-02-12 22:12:46,351 [ERROR|main$] Incorrect repository argument
2018-02-12 22:12:46,351 [INFO|main$] Please select 'encode' or 'tcga'
```

Listing 6.9: Esempio di errore per il parametro della 'repository'

Successivamente il programma controlla la validità del file *xml* e in caso il file di configurazione non risulti corretto rispetto all'*xsd* (vedi Appendice C per i dettagli) viene mostrato un messaggio di errore (Listing 6.10):

```
2018-02-12 21:46:58,705 [INFO|DbHandler$] Start to create the database
2018-02-12 21:47:00,367 [ERROR|main$] Xml file is not valid according the specified schema, check: https://raw.githubusercontent.com/DEIB-GECO/GMQL-Importer/federico/Example/xml/setting.xsd
```

Listing 6.10: Esempio di errore nella lettura del file di configurazione

Superati tutti i controlli sui dati in input il framework verifica, usando le informazioni del database specificato, che le tabelle siano presenti: in caso contrario le crea e viene stampato a video il messaggio indicato nel Listing 6.11.

```
2018-02-12 22:53:38,058 [INFO|DbHandler$] Start to create the database
```

```
2 2018-02-12 22:53:38,440 [INFO|DbHandler$] Table DONORS created
3 2018-02-12 22:53:38,470 [INFO|DbHandler$] Table BIOSAMPLES
    created
4 2018-02-12 22:53:38,503 [INFO|DbHandler$] Table REPLICATES
    created
5 2018-02-12 22:53:38,530 [INFO|DbHandler$] Table EXPERIMENTSTYPE
     created
6 2018-02-12 22:53:38,554 [INFO|DbHandler$] Table PROJECTS
    created
7 2018-02-12 22:53:38,584 [INFO|DbHandler$] Table CONTAINERS
    created
8 2018-02-12 22:53:38,610 [INFO|DbHandler$] Table CASES created
9 2018-02-12 22:53:38,633 [INFO|DbHandler$] Table ITEMS created
10 2018-02-12 22:53:38,699 [INFO|DbHandler$] Table REPLICATESITEMS
     created
11 2018-02-12 22:53:38,726 [INFO|DbHandler$] Table CASESITEMS
    created
12 2018-02-12 22:53:38,752 [INFO|DbHandler$] Table DERIVEDFROM
    created
```

Listing 6.11: Creazione delle tabelle

Una volta create le tabelle, il programma inizia ad analizzare tutti i file che soddisfano la *regex* della sorgente indicata. A video compare un messaggio di **INFO** che mostra lo stato del file preso in considerazione: se lo stato risulta *archived*, o nel caso in cui non sia possibile definire lo stato del file, il framework ignora il suo contenuto e passa ad analizzare il file seguente (vedi Listing 6.12).

```
1 2018-02-02 14:20:20,566 [INFO|main$] Start to read /home/nacho/
    GMQL-Importer/workingDirectory/HG19_ENCODE/narrowPeak/
    Transformations/ENCFF059VEO.bed.meta.json
2 2018-02-02 14:20:20,580 [INFO|main$] File status archived, go
    to next file
3 2018-02-02 14:20:20,580 [INFO|main$] Start to read /home/nacho/
    GMQL-Importer/workingDirectory/HG19_ENCODE/narrowPeak/
    Transformations
4 2018-02-02 14:20:20,769 [INFO|main$] File status released
```

Listing 6.12: Lettura dei file nella working directory indicata

Al contrario, in caso lo stato del file risulti *released*, il framework inizia la routine di lettura e, per mezzo del file di configurazione, recupera i dati necessari per la popolazione delle istanze delle tabelle. Nell'eventualità in cui uno dei mapping non sia trovato, un messaggio di **WARNING** viene mostrato a video (vedi Listing 6.13).

```
2018-02-02 14:20:22,769 [INFO|main$] File status released,
    start populate table
2018-02-02 14:20:22,776 [WARN|main$] SourceKey not found in
    operation<BIOSAMPLES,
    replicates__1__library__biosample__health_status, isHealthy,
     DEFAULT>
2018-02-02 14:20:22,776 [WARN|main$] SourceKey in found for
    operation<BIOSAMPLES,
    replicates__1__library__biosample__health_status, disease,
    DEFAULT>
2018-02-02 14:20:22,778 [WARN|main$] SourceKey does in find for
     operation<EXPERIMENTSTYPE, replicates__1__antibody__url,
    antibody, CONCAT>
```

Listing 6.13: Esempio di data file Encode

Eventuali errori di lettura del file di input, nel caso in cui il caso sia già noto, vengono loggati con uno specifico messaggio (Listing 6.14), mentre viene stampato a video un messaggio generico in qualsiasi altro caso di errore (Listing 6.15).

```
2018-02-02 14:20:20,566 [ERROR|main$]
    ArrayIndexOutOfBoundsException file with path /home/nacho/
    GMQL-Importer/workingDirectory/HG19_ENCODE/narrowPeak/
    Transformations/ENCFF059VEO.bed.meta.json
```

Listing 6.14: Errore di ArrayIndexOutOfBoundException

```
2018-02-02 14:20:20,566 [ERROR|main$] Unknown File input
    Exception file with path /home/nacho/GMQL-Importer/
    workingDirectory/HG19_ENCODE/narrowPeak/Transformations/
    ENCFF059VEO.bed.meta.json
```

Listing 6.15: Errore generico di lettura del file in input

Se durante l'analisi di un file l'informazione di una qualsiasi delle chiavi univoche delle tabelle (vedi Tabella 4.1) non viene trovata, un messaggio di **WARNING** viene mostrato a video e si passa al contesto del file successivo senza alcun inserimento o aggiornamento.

```
2018-01-30 15:10:57,444 [WARN|EncodeTables] Primary key of it.
    polimi.genomics.importer.ModelDatabase.Encode.Table.
    ExperimentTypeEncode not found
```

Listing 6.16: Errore generato per mancanza della chiave univoca per una tabella nel file in input

A fine esecuzione vengono mostrate a video una serie di statistiche riguardanti le operazioni svolte dal programma (come, per esempio, in Listing 6.17). Tutte le statistiche sono collezionate da contatori e conservate nella classe singleton **Statistics**: in seguito è possibile sfruttare queste

informazioni per effettuare considerazioni sul dataset analizzato o per progettare miglioramenti generali da apportare al programma.

```
2018-02-23 17:49:01,760 INFO [main$] Total time to insert data in DB 2:38:18
2018-02-23 17:49:01,761 INFO [main$] Total analyzed files 12071
2018-02-23 17:49:01,761 INFO [main$] Total released files 9636
2018-02-23 17:49:01,761 INFO [main$] Missing file status (or other metadata) 1967
2018-02-23 17:49:01,761 INFO [main$] Total not inserted released files 0
2018-02-23 17:49:01,761 INFO [main$] Total inserted items 168
2018-02-23 17:49:01,761 INFO [main$] Total updated items 70280
2018-02-23 17:49:01,761 INFO [main$] Constraints violated 0
2018-02-23 17:49:01,761 INFO [main$] Total founded malformation 106
2018-02-23 17:49:01,761 INFO [main$] ArrayIndexOutOfBoundsException file input 0
2018-02-23 17:49:01,761 INFO [main$] UnknownInputException file input 468
```

Listing 6.17: Estratto di statistiche di fine esecuzione

I file di log vengono creati automaticamente ad ogni esecuzione del programma e salvati nella cartella in cui si trova il file eseguito. Per distinguerli, vengono salvati concatenando la modalità di esecuzione, il nome della sorgente a cui l'esecuzione fa riferimento e l'istante temporale di inizio di esecuzione del programma seguendo il pattern "yyyy_MM_dd HH:mm:ss.SSS Z", con estensione *.log*. Un esempio per il nome di un file di log è "IMPORT_ENCODE_2018_02_15 18:00:20.507 +0100.log".

6.9.1 SLF4J e log4j

The Simple Logging Facade for Java (SLF4J) fornisce, attraverso un pattern di tipo façade[5], API per l'implementazione di framework di logging, consentendo all'utente finale di collegare, al momento della distribuzione, il framework di logging desiderato. La separazione dell'API client dal back-end di registrazione riduce l'accoppiamento tra un'applicazione e qualsiasi struttura di registrazione specifica: questa operazione rende più semplice l'integrazione con codice esistente o di terze parti, facilitando inoltre il rilascio di codice all'interno di progetti già esistenti e che hanno già effettuato una scelta per il backend di registrazione dei messaggi. In questo lavoro di tesi, si è utilizzato il framework log4j, una libreria Java sviluppata dalla Apache Software Foundation, che definisce un'architettura di registrazione di logging orientata ai livelli con le seguenti priorità, ordinate in modo crescente, per le tipologie di messaggi: OFF, FATAL,

[5]Letteralmente façade significa "facciata", ed infatti nella programmazione ad oggetti indica un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi.

ERROR, WARN, INFO, DEBUG e TRACE. Oltre alle tipologie di log già citate e utilizzate all'interno del framework (ERROR, WARN e INFO), è possibile sfruttare all'interno del framework anche le altre tipologie di messaggi per i seguenti scopi:

- **OFF**: il livello più alto possibile, viene usato per disattivare i log.
- **FATAL**: segnala un errore importante che causa un prematuro termine dell'esecuzione: ci si aspetta che questo sia visibile immediatamente all'operatore.
- **DEBUG**: usato nella fase di debug del programma. Viene riportato nel file di log.
- **TRACE**: fornisce alcune informazioni dettagliate e ci si aspetta che venga scritto esclusivamente nei file di log.

6.10 Esecuzione del programma in modalità *export*

La seconda modalità di esecuzione del framework è la modalità di *export*: essa permette di esportare il contenuto del database in formato ⟨chiave, valore⟩ dove la chiave e il valore sono divisi da un carattere di tab. Per eseguire il processo è necessario specificare tre input, invece che i quattro previsti dalla modalità di *import*: la modalità di esecuzione, il nome della sorgente e la directory contenente i *data files*. Un esempio di parametri in input per questa modalità è illustrato nel Listing 6.18 (vedi immagine 6.1). Il processo viene iterato su tutti i file contenuti nella directory indicata e su tutte le sue sottocartelle, considerando solamente quelli che rispettano una determinata *regex* che specifica l'estensione dei file da prendere in considerazione: viene quindi estratto il nome del file (non vi è un accesso fisico, ma solo una lettura delle proprietà) che corrisponde al valore di *sourceId* dell'item a cui questo file fa riferimento. A questo punto, un processo di estrazione dei dati, ottenuto per mezzo di interrogazioni del database, popola le istanze delle classi che rappresentano il modello del database. Quando tutte le classi sono state popolate, ha inizio una procedura di trascrizione degli attributi sul *data file* da cui è stato ottenuto il *sourceId* dell'item (in modalità di *append* in coda al file) oppure, in base alla scelta dell'utente, su un nuovo file che avrà nome *sourceId.<regex>.<ext>*. Queste operazioni di esportazione sono definite all'interno delle classi Trait *<TableName>* che descrivono ogni singola tabella definita dal GCM: in questo modo non è necessario ridefinire il processo di esportazione ogni qual volta si debba introdurre la compatibilità di una nuova sorgente nel framework e eventuali modifiche alla procedura di export devono essere definite in un unico punto per tutte le classi che implementano le funzionalità di una determinata tabella. In caso di necessità, le singole classi possono sovrascrivere il metodo di esportazione, oppure definire un comportamento comune per un certo repository: questo metodo deve essere scritto all'interno della classe ***<RepositoryTable>***. L'output di questo processo non crea informazioni aggiuntive, ma uno snapshot del contesto attuale del database nel file di origine. Il risultato può essere utilizzato da altri programmi che necessitano di file di testo per il loro funzionamento, che nel frattempo possano sfruttare una nomenclatura comune per sorgenti eterogenee, rendendo l'implementazione di programmi di interrogazioni dei dati più semplice per l'utente finale e rendendo i file che contengono i dati più leggibili per un essere umano.
L'esecuzione del framework in modalità di *export* è riportata in pseudocodice nell'algoritmo 5.

```
export
encode
/home/federico/Scrivania/_Encode_Download_tsv
```

Listing 6.18: Esempio di parametri in input per la modalità export

Algorithm 5: Modalità di Export dei dati

Data: [DATAFILEPATH]
Result: Estrazione tuple
Andare alla cartella DATAFILEPATH;
Aprire *sourceId*.bed.gz.json file in Downloads folder;
while *Per tutti i file contenuti nella cartella* **do**
  if *Se il file rispetta la regex che specifica l'estensione;*
  then
    Recuperare il *sourceId* escludendo la regex al file*sourceId*.<regex>;
    Recupare l'item dal DB usando $sourceId$;
    Recuperare tutti i dati alle tabelle collegate rispettando l'rdoine;
    if *Se la modalità scelta è in **append*** **then**
      Aprire il file $sourceId_i$. <regex>;
      Incollare in modalità di **append** i dati nel file;
    else
      Creare un nuovo file con nome $sourceId_i$ ed estensione indicata dall'utente;
      Incollare i dati recuperati;

6.10.1 Estrazione dei dati dalle tabelle

Anche per la modalità di *export* esiste un ordine vincolante per l'estrazione dei dati, procedura che ha inizio dalle tabelle più interne per procedere poi verso le tabelle più esterne del GCM. Differentemente dall'ordine definito dalla modalità di *import*, essendo il valore in ingresso sempre e solamente l'attributo *SourceId* dell'item, in questa modalità la tabella di partenza non può essere che *Item*. Estrapolata la tupla e ricavata la chiave primaria, si procede all'interrogazione del database per estrarre i valori di Replicate, Case ed ExperimentType, ognuno ottenuto sfruttando la chiave primaria della tupla ottenuta nello step precedente, rappresentante la chiave esterna di collegamento per queste ultime. Da queste tabelle si ricavano le informazioni delle tabelle a loro associate. Si noti che per ottenere le tuple dei replicati o dei case è obbligatorio passare rispettivamente dalle tabelle ponte ReplicateItem e CaseItem, eseguendo un'operazione di *join*. Essendo DerivedFrom la tabella che definisce l'autoanello su Item, è stato necessario implementare una procedura ricorsiva che percorresse a ritroso la tabella, al fine di elencare tutte le dipendenze dell'item. È opportuno precisare che non vengono esportate tutte le informazioni degli item padri (cioè quelli da cui l'item di inizio procedura dipende direttamente) e dei suoi antenati (item che dipendono dai file padre dell'item di inizio procedura, e così via), ma viene esportato solamente il valore di *SourceId*, in modo che sia possibile ricercare l'item desiderato semplicemente mediante una query sulla base di dati. La riga inserita nel file è quindi com-

posta dall'attributo che ha nome PREFISSO__NOMETABELLA__NOMEATTRIBUTO (il doppio carattere di underscore indica il carattere separatore), con singolo carattere di underscore che va a sostituire eventuali spazi contenuti nel nome dell'attributo. All'attributo è associato il valore corrispondente nella tabella di riferimento. Sia il prefisso sia il carattere separatore fra le varie sezioni dell'attributo possono essere scelti dall'utente attraverso il file di *application.conf* del framework, anche se come default il valore separatore è il doppio underscore, mentre il prefisso è *integrated*. Un esempio di coppia attributo-valore, riferita all'entità *Project* e colonna ProgramName, usando i valori di default può essere: "integrated__project__program_name Encode".
Nel caso della sorgente ENCODE le informazioni estratte dalle tabelle Replicate, Biosample e Donor contengono il riferimento al replicato biologico, in quanto è possibile che in un esperimento siano coinvolti alcuni di questi elementi.

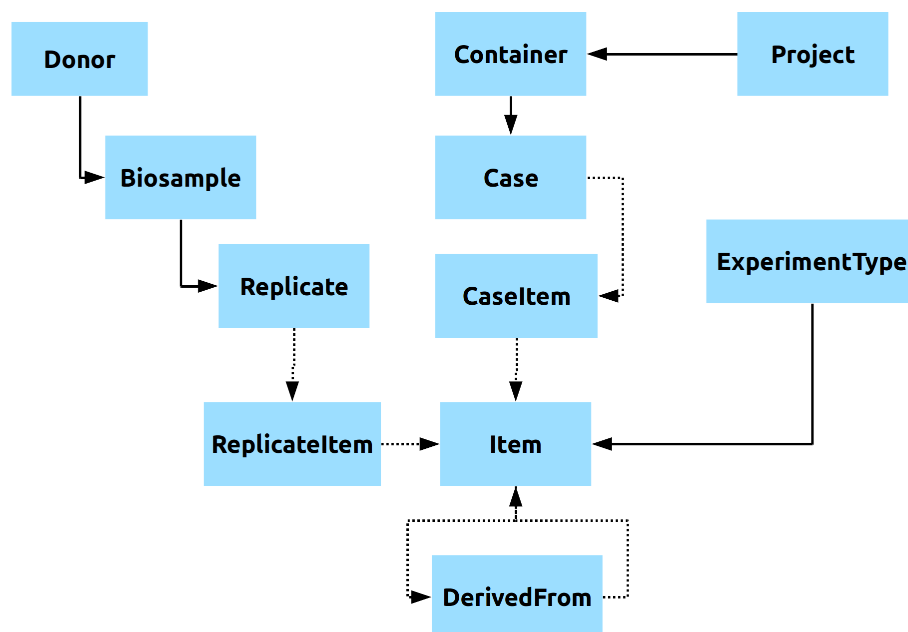


Figura 6.7: Dipendenze delle tabelle nella procedura di export

6.10.2 Possibili messaggi di log

Come per la modalità di *import* dei dati, anche nella modalità di *export* l'utente può seguire l'andamento dell'esecuzione per mezzo di messaggi di log mostrati a video che saranno successivamente salvati in un file quando l'operazione sarà terminata.
Una volta verificata la presenza del database, il framework controlla eventuali errori nei parametri in ingresso, guidando l'utente per una corretta impostazione, Listing 6.19

```
1 2018-03-03 17:29:59,113 [INFO|it.polimi.genomics.importer.
    RemoteDatabase.DbHandler$|DbHandler$] Check database
```

```
2 2018-03-03 17:29:59,131 [ERROR|main$] Incorrect number of
    arguments
3 2018-03-03 17:29:59,131 [INFO|main$] GMQLImporter help; in
    order to export Data from Database to file Run as arguments:
     EXPORT, repository_ref and file_folder
```

Listing 6.19: Esempio di errore nei parametri in ingresso per la modalità di export

Successivamente alla verifica dei parametri di input, si prosegue con la lettura di tutti i file nella cartella che rispettano la regex che specifica l'estensione dei file per la sorgente indicata. Un messaggio informa dell'inizio della procedura di *export* per il singolo file e dell'eventuale successo della procedura Listing 6.20.

```
1 2018-03-03 17:51:04,648 [INFO|FromDbToTsv] Start to read /home/
    federico/Scrivania/_Encode_Download_tsv/HG19_ENCODE/
    broadPeak/Transformations/ENCFF900EEO.bed.meta.json file
2 2018-03-03 17:51:05,142 [INFO|FromDbToTsv] File /home/federico/
    Scrivania/_Encode_Download_tsv/HG19_ENCODE/broadPeak/
    Transformations/ENCFF900EEO.bed.meta.json correctly exported
```

Listing 6.20: Esempio di una corretta esportazione

Nel caso in cui non esista la tupla cercata nel database, oppure venga riscontrato un qualsiasi altro errore durante la procedura, un messaggio di **ERROR** avverte l'utente dell'anomalia e il contesto del framework passa a file successivo Listing 6.21.

```
1 2018-03-03 17:37:55,452 [INFO|FromDbToTsv] Start to read /home/
    federico/Scrivania/_Encode_Download_tsv/HG19_ENCODE/
    broadPeak/Transformations/ENCFF201NAO.bed.meta.json file
2 2018-03-03 17:37:55,532 [ERROR|Item] Item: Not tuple found
    Exception
3 2018-03-03 17:37:55,587 [ERROR|FromDbToTsv] Some error in
    FromDbToTsv process, go to next file
4 2018-03-03 17:37:55,532 [ERROR|it.polimi.genomics.importer.
    ModelDatabase.Encode.Table.ItemEncode|Item] Item: Not tuple
    found Exception
5 2018-03-03 17:37:55,587 [ERROR|it.polimi.genomics.importer.
    ModelDatabase.FromDbToTsv|FromDbToTsv] Some error in
    FromDbToTsv process, go to next file
```

Listing 6.21: Esempio di errore di tupla non trovata e pasaggio al file successivo

A fine processo, un log di statistica informa quanti export sono andati a buon fine e quanti sono terminati andando in errore Listing 6.22

```
1 2018-03-04 15:37:13,371 [INFO|main] Total time for the write
    info in TSV file 0:0:3
2 2018-03-04 15:37:13,372 [INFO|main] Total file analized 3
3 2018-03-04 15:37:13,372 [INFO|main] File correctly exported 3
```

```
4 2018-03-04 15:37:13,372 [INFO|main] File exported with Error 0
```
Listing 6.22: Esempio di Statistiche nella modalità di export

6.11 Application Configuration file

L'Application Configuration File è un file che permette di modificare alcuni comportamenti del framework, sia nel contesto della modalità di *import* sia in quello di *export* dei dati. Questa scelta è nata dall'esigenza di utilizzare il framework in contesti diversi, ognuno con diverse peculiarità, nell'ambito del progetto GeCo. Introducendo questo file, si è raggiunta una versatilità tale da rendere usufruibile il programma in ogni contesto. Nel dettaglio, il file si compone di tre parti configurabili:

- **database** è la sezione che specifica il database da utilizzare sia per l'importazione sia per l'esportazione dei dati. È composto a sua volta da quattro parametri che specificano rispettivamente l'url, l'username, la password per accedere al database e il driver che Slick utilizza per comunicare con la base di dati.

- **import** è la sezione che indica la configurazione per la modalità di integrazione dei dati. In questa sezione si può specificare tramite il parametro derived_item se attivare o meno la procedura di import degli item non processati. Attraverso il parametro constraints_activated impostato a `true` si attiva invece la modalità di controllo *forte* dei vincoli e delle regole presentate nella Sezione 4.4; contrariamente, è considerata attiva la modalità di controllo *debole*. multiple_value_concatenation definisce la stringa utilizzata per concatenare valori diversi che nel *data file* sono mappati con medesimi attributi. Un esempio è fornito dall'attributo *feature* dell'entità **ExperimentType** per la sorgente ENCODE dove, nel caso non si riesca a trovare alcuna corrispondenza nel *data file* per la chiave file__replicate__experiment__target__investigated_as, si considera la chiave target__investigated_as che, come possiamo notare nel Listing 6.24, nel file in input può corrispondere a diversi valori. L'attributo booleano support_table_insert specifica se le tabelle di supporto alle ontologie (vedi Sezione 6.12) devono essere inserite o meno durante l'esecuzione del framework.
  L'attributo method_character_separation indica il carattere separatore da utilizzare nel file di configurazione per gli attributi SUB_CHARACTER, NEW_CHARACTER e SUB_CHARACTER. Inoltre, è possibile attivare/disattivare particolari regole attraverso l'attributo rules: questa funzionalità è stata utilizzata solo in fase di sviluppo del codice e analisi dei dati per monitorare l'impatto delle regole nel contesto del framework; durante la normale importazione dati tutte le voci contenute nell'attributo rules devono essere impostate a `true`.

- **export** è la sezione in cui si specificano i parametri per l'esecuzione del framework nella modalità di esportazione dei dati. Si possono specificare, tramite il parametro prefix e separation, rispettivamente il prefisso che precederà tutti i dati esportati dal database (vedere Sezione 6.10.1) e il carattere separatore fra le sezioni di PREFISSO, NOMETABELLA ed ATTRIBUTO. Inoltre, settando il parametro NEWFILE a `false`, il framework verrà eseguito in modalità di *append* ai file già esistenti, mentre settandolo a `true` verrà creato

un nuovo file con nome uguale all'attributo *sourceId* dell'item di cui stiamo esportando le informazioni e con estensione specificata dal parametro extension.

Di default sono attive le configurazioni specificate nel Listing 6.23. In modalità di *import* è attiva la routine di ricerca dei file derivati e il controllo delle dipendenze è eseguito in senso *debole*, mentre nella modalità di *export* il prefisso è la stringa "integrated" con il doppio carattere di underscore come separatore delle sezioni; i dati esportati vengono inseriti in append ai file presenti nella directory indicata come parametro in input al framework. È possibile modificare questi parametri di default passando al framework, al momento dell'esecuzione del programma, il nuovo file di configurazione, tramite il parametro opzionale `Dconfig.file` indicante la locazione del nuovo file da prendere in considerazione. Il nuovo file deve specificare totalmente tutti gli attributi, compresi quelli di default, anche se si volesse ridefinirne solo una parte: infatti, in caso essi non siano interamente riportati, il framework terminerebbe l'esecuzione del programma con un errore.

```
database {
 url = "jdbc:postgresql://localhost/geco"
 username = "****"
 password = "****"
 driver = "org.postgresql.Driver"
}

import {
 derived_item = true
 constraints_activated = false
 multiple_value_concatenation = ", "
 support_table_insert = true
 method_character_separation = "*"
 rules {
  is_healthy = true
  type = true
 }
}

export {
 separation = "__"
 prefix = "integrated"
 newfile = false
 extension = "txt"
}
```

Listing 6.23: Parametri di configurazione di Default

```
target__investigated_as histone
target__investigated_as histone modification
```

Tabella 6.5: Tabella delle ontologie

Attribute	Descrizione	Tipo	Nullable
table_id	Chiave primaria della tupla a cui l'ontologia fa riferimento	Int	No
table_name	Nome della tabella a cui l'ontologia fa riferimento	String	No
table_column	Nome dell'attributo a cui l'ontologia fa riferimento	String	No
original_key	Valore di mapping contenuto nel file di configurazione	String	No
original_value	Valore contenuto nella *tabella table_name*, della colonna *table_column* per la tupla con chiave primaria *table_id*	String	No
ontological_code	Codice ontologico	String	Sì

```
3 target__investigated_as narrow histone mark
```

Listing 6.24: Estratto di data file per la sorgente ENCODE

6.12 Tabella per il supporto alle ontologie

In previsione a futuri sviluppi ed implementazioni di nuove funzionalità all'interno del framework, è stata progettata, nel contesto di questa tesi, la specifica di una tabella funzionale al supporto delle ontologie. Nel dettaglio, è stata definita la tabella descritta in 6.5 con lo scopo di specificare tutti i codici ontologici per i valori inseriti nel database. Attualmente la tabella supporta l'importazione delle ontologie per gli attributi *cellLine* e *tissue* dell'entità **Biosample** e *technique* dell'entità **ExperimentType** esclusivamente per quanto riguarda la sorgente ENCODE. I codici ontologici che si intendono importare sono specificati nel file di configurazione attraverso il metodo *ONTOLOGY* (esempio nel Listing 6.25): questo speciale metodo va a popolare la variabile specificata nella `global_key`, un attributo esistente esclusivamente all'interno della classe **Biosample** o **ExperimenType** di ENCODE (a seconda dell'ontologia che stiamo importando), e del quale non vi è alcuna corrispondenza nella tabella che ne specifica l'entità; esso viene sfruttato per specificare la chiave di importazione e il codice ontologico dell'attributo desiderato. Nel dettaglio, il metodo concatena, dividendoli con un carattere speciale, il valore della `source_key`, definito nel file di configurazione, con il valore corrispondente trovato nel data file: in questo modo è possibile importare due valori con un unico mapping. Questi valori popolano rispettivamente la colonna *original_key* e *ontological_code* della tabella. Dovendo specificare nella tabella anche la chiave primaria della tupla a cui l'ontologia fa riferimento, è necessario che l'inserimento dell'ontologia sia successivo all'inserimento della tupla relazionata. Non è possibile specificare il valore dell'attributo *table_column*: questa informazione è esplicitata con una costante nel codice. Nel caso dell'entità **Biosample**, l'ontologia potrebbe riferirsi sia alla colonna *tissue* che a *cellLine*: sarà il framework a determinare il valore di *table_column*, controllando nella classe quale dei due attributi non ha valore nullo, sulla base della regola secondo cui solamente uno di essi può assumere valore non nullo. Anche per la tabella per il supporto alle ontologie è stata implementata una procedura di inserimento/aggiornamento identica a quella prevista per le tabelle delle entità 6.3. La chiave primaria della tabella è specificata dalla tripla *table_id*, *table_name*, *table_column*, mentre la colonna *ontological_code* può assumere valore nullo.

```
1 <mapping method="ONTOLOGY">
2   <source_key>assay_term_id</source_key>
```

```
3     <global_key>ontologicalcode</global_key>
4 </mapping>
```
Listing 6.25: Esempio di mapping per l'importazione di un ontologia

6.13 Complessità temporale

Analizzando l'algoritmo presentato nella Sezione 6.5, è possibile determinarne la complessità temporale. Consideriamo l'esecuzione del programma per la sorgente TCGA per un singolo file e *n* regole di mapping contenute nel file di configurazione. Le operazioni che si susseguono, in ordine di esecuzione, sono:

- la creazione, per ogni regola di mapping, dell'operazione ad essa associata e la concatenazione della stessa alla catena di operazioni;
- la creazione della struttura che permette di mappare le coppie $\langle$chiave, valore$\rangle$ del file in ingresso. Questo processo può essere considerato come un'operazione atomica, in quanto sfrutta operazioni di librerie di sistema;
- l'esecuzione di una mappatura dei valori per ogni operazione contenuta nella catena delle operazioni in modo da popolare le varie classi;
- l'esecuzione di un numero di operazioni di controllo della presenza della tupla e di inserimento/aggiornamento pari al numero delle tabelle definite (considerando anche le tabelle ponte).

Si ottiene la seguente funzione di complessità:

$$f(n) = 2n + 1 + n + 2 \cdot 11 = 3n + 23 \quad (6.1)$$

Questa funzione dimostra che la complessità è lineare rispetto al numero di mappature contenute nel file di configurazione; per TCGA, considerando che ogni file in ingresso determina l'inserimento di una ed una sola tupla per ogni tabella, il caso pessimo coincide con il caso ottimo e si ha $f(n) \in O(n)$.
Per ENCODE il caso ottimo coincide con il caso base di TCGA, in quanto vengono inseriti rispettivamente un solo donatore, biosample e replicato e dove ogni item non ha alcun antenato. Il caso ottimo in ENCODE è molto raro, genericamente in questa sorgente ad ogni item radice sono legati diversi altri item e vi è la possibilità di inserire molteplici donatori, biosample e replicati per ogni file: avendo sfruttato una Ricerca in Profondità, come spiegato in 6.7.1, per la ricerca degli item collegati all'item *root* vi è la necessità di aggiungere il numero di operazioni introdotte dalla visita del Directed Acyclic Graph e le relative azioni di inserimento o aggiornamento dei nuovi item. La complessità per la visita di un DAG, usando un approccio Depth First Search, è $O(|V| + |E|)$, dove V indica il numero di nodi (nel nostro caso gli item inseriti) ed E indica gli archi. Per il caso pessimo dobbiamo considerare un DAG con il massimo numero di archi, che sono pari a $|V| \cdot |V - 1|$ (esempio in Figura 6.8), ottenendo quindi una complessità

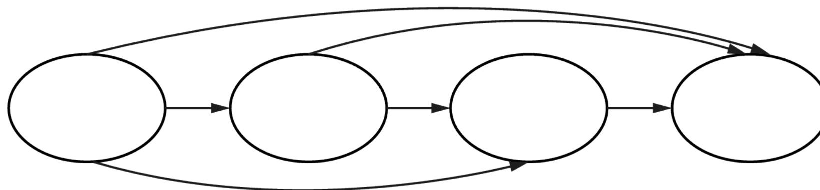


Figura 6.8: Rappresentazione di un DAG con massimo numero di archi.

quadratica rispetto al numero di item. Nella maggior parte dei casi reali analizzati il grafo degenera in un albero, cioè esiste un solo percorso per raggiungere ciascun vertice, e quando questo non accade vi è effettivamente un numero limitato di nodi per cui esistono più cammini per raggiungerlo: questo riduce la complessità nel caso reale. Le operazioni da svolgere per ogni nodo V non si limitano al popolamento degli attributi dell'entità e dell'aggiornamento/inserimento dell'item nella base di dati, ma vi è la necessità di eseguire le operazioni di controllo della presenza e inserimento/aggiornamento anche per le tabelle di collegamento **CaseItem** e **ReplicateItem**: si ottiene una complessità finale pari a $O(|V|(8 + 2 * 3) + |E|)$, dove $|V|$ assume mediamente un valore di 7,12, come indicato nella Tabella 6.6. Si noti inoltre come il numero medio di donatori, biosample e replicati sia molto vicino a uno ma nello stesso tempo possiamo sia evidente un sensibile incremento al numero di accessi al database.

Assembly	**File**	**Donatori**	**Biosample**	**Replicati**	**Item**
HG19	10534	15207	15207	15732	73254
GRCh38	10091	14627	14627	15292	73582
Somma	20625	29834	29834	31024	146836
Media operazioni per file		1,45	1,45	1,50	7,12

Tabella 6.6: Riepilogo del numero di inserimenti/aggiornamenti per file in ENCODE.

6.14 Tempi di esecuzione e performance

Considerando che il processo di integrazione dei dati implementato sfrutta l'approccio **data warehouse**, è necessaria una continua e periodica importazione dei metadati per poter mantenere il database aggiornato con le sorgenti. Non disponendo di alcun sistema per la verifica di potenziali cambiamenti dei dati nelle sorgenti, l'importazione completa dei dati attraverso il processo ETL risulta essere attualmente l'unica modalità applicabile per l'aggiornamento delle informazioni contenute nel repository condiviso.
In questa sezione, sono indicate ed analizzate le tempistiche di esecuzione per ognuna delle tre fasi del processo ETL: queste statistiche sono funzionali all'analisi di eventuali *bottleneck* all'interno del framework e rendono possibile la ricerca di soluzioni per il miglioramento delle

attuali prestazioni.
Le tabelle presentate sono il risultato di molteplici esecuzioni (Appendice E per i dati completi), svolte in più giornate: distribuire gli esperimenti uniformemente nel tempo è la modalità migliore per ottenere un risultato medio in cui siano considerate anche esecuzioni contemporanee di altre attività, secondo il normale funzionamento di un server.
Le caratteristiche della macchina con cui sono stati effettuati i test sono:

- Linux version 3.2.0-4-amd64 Debian 3.2.51-1
- Intel® Xeon® Processor with CPU E5-2650 at 2.00 GHz, six cores
- RAM RAM of 387 GB
- 32 thread
- Disco raid 5, 8 dischi da 2TB

6.14.1 Performance sull'inserimento dei dati

Alcune delle statistiche mostrate al termine dell'esecuzione del framework sono adibite alla registrazione dei tempi necessari per l'esecuzione dei tre step del processo ETL. In accordo con le previsioni effettuate, la maggior parte del tempo viene dedicata al processo di *load*, cioè al caricamento dei dati elaborati sul repository condiviso, essendo questo il processo che richiede una comunicazione maggiore con il disco fisso. Inoltre, le query di ricerca, che determinano se per ogni singola tupla sia necessaria un'operazione di inserimento o di aggiornamento, vengono eseguite sfruttando la chiave univoca che, essendo sempre di tipo stringa (per ExperimentType è una tripla di stringhe), determina un inevitabile degrado di performance, in quanto, in un database relazionale, le query non sono ottimizzate per questa tipologia di ricerche.
Nelle Tabelle 6.7 6.8 sono riassunte le statistiche delle performance dell'esecuzione del framework per le sorgenti ENCODE e TCGA in contesti identici, cioè disabilitando per ENCODE il supporto alle ontologie e ai file derivati: si può osservare che in entrambi i casi il processo più oneroso è il *load* che impiega più del 90% del tempo dell'intera esecuzione. Confrontando le tempistiche tra le due sorgenti, è emerso che per ENCODE il processo di *load* risulta più oneroso rispetto a TCGA in quanto, generalmente, vi è la possibilità di inserire più replicati, biosample e donatori per ogni file, con conseguente diminuzione della velocità di inserimento di item per secondo; al contrario, per TCGA risulta più oneroso il processo di trasformazione, rispetto al caso di ENCODE: questo risultato è giustificabile considerando che per determinare il valore dell'attributo *sourceSite* dell'entità **Case** è necessario l'accesso ad una tabella di supporto contenuta nel database 4.5.
La tabella 6.9 espone le statistiche riguardo le tempistiche per l'importazione dei metadati dalla sorgente ENCODE nel caso in cui siano attivate le opzioni di importazione delle ontologie e dei file derivati: è evidente che le performance di estrazione e trasformazione dei dati sono molto simili alla situazione precedente, in quanto il numero dei file è identico e non è stata introdotta alcuna operazione aggiuntiva: questa costanza, a fronte di un aumento del carico delle operazioni di *load* rispetto al caso precedente, evidenzia una sostanziale diminuzione dell'impatto di questi

Set	*derived_from*	*support_table_insert*	Item Inserted	Item Updated	Total Item	Extract	Transform	Load	Total	Item/sec
HG_19	FALSO	FALSO	10534	0	10534	0.00.51	0.01.03	0.38.17	0.40.11	4,37
GRCh38	FALSO	FALSO	10091	0	10091	0.01.10	0.00.46	0.39.45	0.41.41	4,04
Somma			20625	0	20625	0.02.31	0.01.49	1.18.02	1.21.52	4,21
Percentuale						2,46%	2,21%	95,33%		0,23

Tabella 6.7: Tabella riassuntiva per inserimenti dei dati in ENCODE. Le ultime due righe dell'ultima colonna indicano media e deviazione standard.

processi in termini assoluti sulle tempistiche di esecuzione del framework a discapito, ancora una volta, del processo di caricamento, che risulta prolungato principalmente per la necessità di inserire nella base di dati i file derivati. In questo caso, il bilancio complessivo della fase di *load* arriva addirittura al 98% ma è altrettanto evidente come il numero di item per secondo sia molto maggiore nel secondo caso, in cui è possibile, a differenza del primo, l'inserimento di più item attraverso la lettura in input di un unico file: il peso degli inserimenti di tutte le altre tabelle viene suddiviso su più item.
In generale, dalle statistiche presentate emerge che l'unico processo per cui potrebbero essere necessarie modifiche di miglioramento delle performance è il processo di caricamento dei dati. Di seguito, alcune soluzioni proposte:

- Dare in input al framework, per successivi aggiornamenti, solo i dati che hanno subito modifiche: in questo modo, il framework non dovrebbe subire alcuna modifica dal punto di vista del suo funzionamento.
- Modificare la funzionalità di verifica della presenza della tupla nel database in modo da non controllare esclusivamente la presenza della tupla tramite la chiave univoca ma da restituire anche un valore che permetta di determinare se la tupla necessita di un effettivo aggiornamento. Questa funzionalità può essere implementata in due modi: direttamente nel database attraverso una query, a cui vengono trasmessi tutti i dati di una tabella, che, nel processo di recupero della tupla ne verifica anche l'omogeneità e, nel caso in cui questa non sia riscontrata, effettua un'operazione di update. Alternativamente, si può effettuare il controllo dell'eventuale omogeneità dei dati direttamente nel codice, facendo tornare dalla query, data la chiave univoca, l'intera tupla, in modo che sia il codice del framework ad effettuare questa verifica. Nonostante la seconda opzione comporti una modifica sostanziale del codice e sia caratterizzata da una più difficile manutenibilità, ma garantirebbe probabilmente migliori prestazioni.

Dalle statistiche presentate si può intuire come la parallelizzazione dei tre processi ETL, per l'esecuzione contemporanea di più file ognuno in una differente sezione, sia totalmente inutile, in quanto la quantità di tempo richiesto dal processo di caricamento è molto superiore alle altre: il tempo risparmiato sarebbe di poco inferiore al 5% del totale anche nella migliore delle aspettative.

6.14.2 Differenza delle operazioni con un database popolato

Dalla tabella 6.10 possiamo verificare che le operazioni svolte su un database popolato sono leggermente più onerose rispetto alle stesse operazioni svolte su un database provvisto di un numero

Set	*derived_from*	*support_table_insert*	Item Inserted	Item Updated	Total Item	Extract	Transform	Load	Total	Item/sec
copy_number_segment	-	-	22374	0	22374	0.02.31	0.03.02	0.58.10	1.03.25	5,83
isoform_expression_quantification	-	-	10999	0	10999	0.01.35	0.01.35	0.32.16	0.35.26	5,18
masked_somatic_mutation	-	-	10187	0	10187	0.01.40	0.01.29	0.29.22	0.32.31	5,22
mirna_expression_quantification	-	-	10947	0	10947	0.00.48	0.01.32	0.31.02	0.33.23	5,47
gene_expression_quantification	-	-	11091	0	11091	0.01.36	0.01.32	0.31.23	0.34.31	5,36
masked_copy_number_segment	-	-	22375	0	22375	0.03.13	0.03.23	1.10.26	1.17.02	4,95
methylation_beta_value	-	-	12218	0	12218	0.01.52	0.02.09	0.44.17	0.48.19	5,45
Somma			100191	0	100191	0.10.45	0.11.41	3.58.46	4.21.12	6,39
Percentuale						4,11%	4,47%	91,41%		0,27

Tabella 6.8: Tabella riassuntiva per inserimenti dei dati in TCGA. Le ultime due righe dell'ultima colonna indicano media e deviazione standard.

Set	*derived_from*	*support_table_insert*	File Inserted	Item Inserted	Item Updated	Total Item	Extract	Transform	Load	Total	Item/sec
HG_19	VERO	VERO	10534	16884	56370	73254	0.01.48	0.00.55	2.28.55	2.31.37	8,06
GRCh38	VERO	VERO	10091	24049	49533	73582	0.01.45	0.00.46	2.09.40	2.12.12	9,33
Somma				40933	105903	146836	0.03.33	0.01.41	4.38.35	4.43.49	8,70
Percentuale							1,25%	0,60%	98,16%		0,90

Tabella 6.9: Tabella riassuntiva per inserimenti dei dati in ENCODE con opzione di file derivati e ontologie. Le ultime due righe dell'ultima colonna indicano media e deviazione standard.

minore o completamente privo di dati. Questa considerazione è facilmente intuibile se si considera che per ogni file è necessario verificare la mancanza o la presenza della tupla all'interno del database, al fine di discriminare un'operazione di insert oppure di update: ne consegue che le query di ricerca saranno più onerose all'aumentare dei dati all'interno del database.
La tabella 6.10 indica il caso estremo di inserimento di un dataset all'interno di un database completamente vuoto rispetto all'aggiornamento del medesimo dataset nelle condizioni di completo inserimento dei dati a disposizione.

6.14.3 Differenza fra inserimento e update

Dalla tabella 6.11 è osservabile come, in termini di performance, le operazioni di aggiornamento siano lievemente più onerose rispetto alle operazioni di inserimento. Infatti, per quanto riguarda un inserimento, una volta terminata la query di ricerca e determinato che la tupla non sia presente all'interno del database, è possibile procedere nell'immediato con un inserimento dei dati; contrariamente, nel caso di update, è invece necessario, nel contesto dell'operazione, rieseguire la ricerca della tupla per poter aggiornare le sue informazioni. Nel caso di update, il degrado delle prestazioni è comunque contenuto dal fatto che medesime operazioni, svolte consecutivamente, sono ottimizzate dal dbms, che limita il decremento delle prestazioni all'incirca al 5%.

Database popolato	Item Inserted	Item Updated	Total Item	Extract	Transform	Load	Total	Item/sec
Sì	22375	0	22375	0.06.40	0.03.02	1.01.30	1.11.12	5,24
No	22375	0	22375	0.01.09	0.03.01	0.58.46	1.02.56	5,93
Differenza						4,44%		

Tabella 6.10: Statistiche dell'inserimento del dataset masked_copy_number_segment GRCh38 di TCGA nel contesto di database vuoto e con dati.

Azione	Item Inserted	Item Updated	Total Item	Extract	Transform	Load	Total	Item/sec
Insert	22375	0	22375	0.02.13	0.03.02	0.58.10	1.03.25	5,83
Update	0	22375	22375	0.01.10	0.03.04	1.01.48	1.06.02	5,65
Differenza						5,89%		

Tabella 6.11: Statistiche dell'inserimento e aggiornamento del dataset masked_copy_number_segment GRCh38 di TCGA.

6.14.4 Esecuzioni su macchine differenti

Una maggiore potenza di calcolo non incide sulle prestazioni del framework: infatti, si può verificare dalla Tabella 6.12 come esecuzioni su macchine profondamente differenti abbiano portato alle medesime prestazioni. Come si può notare, non è stato riscontrato un aumento delle prestazioni per alcun settore: questo dimostra che le operazioni non sono onerose né in termini di CPU né in termini di memoria RAM, ma si necessita solamente di una maggiore velocità di accesso al disco. Le caratteristiche della macchina locale su cui sono stati effettuati i test sono:

- Laptop Dell Latitude E5550 64 bits
- Sistema operativo Ubuntu 16.04LTS
- Microprocessore Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
- RAM 8GB SODIMM DDR3 Synchronous 1600 MHz Kingston
- Disco Samsung SSD 850 da 250GB
- Scheda video Intel® HD Graphics 5500

Macchina	Total Item	Extract	Transform	Load	Total	Item/sec
Personale	3406	0.00.08	0.00.28	0.09.20	0.09.57	2,28
Server	3406	0.00.11	0.00.28	0.09.36	0.10.15	2,22

Tabella 6.12: Statistiche dell'inserimento del dataset methylation_beta_value GRCh38 di TCGA su macchine diverse a confronto.

Capitolo 7

Estendibilità del framework

Uno degli scopi principali del progetto è la realizzazione di un framework facilmente estendibile per quanto riguarda l'introduzione di futuri cambiamenti nelle regole di mapping, di nuovi repository o di modifche nel formato dei file di configurazione o di input.

7.1 Estendibilità nelle funzioni di inserimento e regole di mapping

Come già dichiarato nella sezione 6.4.3, sono attualmente disponibili diverse funzioni di inserimento per i valori in ingresso implementati nel modulo **InsertMethod**. Per sviluppare una nuova funzione di input è sufficiente definire un nuovo metodo nella classe **InsertMethod** e associarlo ad una parola chiave che dovrà essere richiamata nel file di configurazione: la parola chiave non deve contenere il simbolo di trattino, in quanto è il simbolo utilizzato per definire molteplici metodi nel contesto di una singola mappatura. Il nuovo metodo sviluppato deve, nel caso fosse necessario, utilizzare correttamente gli attributi SUB_CHARACTER, NEW_CHARACTER e SUB_CHARACTER. Il framework è già predisposto alla composizione multipla dei nuovi metodi. In Scala, implementando le funzionalità tipiche dei linguaggi funzionali, è possibile dichiarare un metodo che ritorni una funzione come parametro: ad esempio, il metodo *selectInsertionMethod* accetta in ingresso i parametri *sourceKey*, *globalKey* e *method* letti dal file di configurazione e restituisce in uscita una funzione che accetta in ingresso il parametro corrente dell'attributo (*actualParam*) e il nuovo parametro (*newParam*) e, basandosi sulla parola chiave di *method*, restituisce la funzione desiderata. In caso non fosse disponibile il metodo scelto, viene lanciata un'eccezione che sarà mostrata a video sfruttando un messaggio di WARNING.
Per inserire una nuova regola di mapping è sufficiente definire all'interno del file di configurazione un nuovo tag **<mapping>**, racchiuso nella tabella di riferimento, utilizzando la struttura specificata nella sezione 6.4.2: inoltre, è necessario specificare con *sourceKey* la chiave del file data in input e con *globalKey* l'attributo di riferimento da popolare nella tabella di target, nonché gli eventuali attributi della sezione di mapping, nel caso fossero necessari.

7.2 Introduzione di un nuova sorgente

Nel caso si volesse che il framework supportasse una nuova sorgente di dati, sarebbe sufficiente effettuare alcuni semplici passaggi per realizzare un supporto efficace. Di seguito sono riportate tutte le fasi che devono essere svolte a questo fine e che consentono di sfruttare a pieno l'architettura implementata e ottimizzare il riutilizzo del codice.
Si ipotizza l'introduzione di un nuovo repository con nome *Repo*, del quale si dispone del *data file* e del file di configurazione. Il primo passaggio richiede la definizione di una classe astratta con nome ***RepoTable***: al suo interno saranno implementati tutti i metodi comuni alle tabelle del repository. Potrebbe non essere necessario implementare alcun metodo in questa classe, ma la sua definizione è in ogni caso vantaggiosa, in quanto contribuisce a rendere il codice più flessibile per evenienze future. La classe astratta ***RepoTable*** può anche essere utilizzata per implementare il pattern di *dependency injection* in riferimento ad un determinato repository, nel caso in cui più tabelle dovessero condividere la stessa istanza di una classe. Successivamente è necessario definire una classe vera e propria per ogni tabella del database in modo che ognuna di esse implementi sia *RepoTable* che la classe Trait di riferimento alla specifica tabella: i nomi delle tabelle dovranno essere **DonorRepo**, **BioSampleRepo**, **ReplicateRepo** ecc. In queste classi dovrà essere implementato unicamente il metodo *setParameter* con le relative regole di dipendenza intratabellari 4.4 e le funzioni di trasformazione 4.5: nel caso ve ne fosse la necessità, si possono sovrascrivere i metodi implementati nelle sottoclassi per gestire casi particolari della sorgente. Infine, è essenziale la definizione della classe "contenitore" delle tabelle, il cui nome sarà **RepoTables** che dovrà semplicemente definire il metodo *getNewTable* il cui compito è restituire, in base al valore a cui ogni tabella è associata, una nuova istanza della tabella del repository a cui fa riferimento.
Svolti correttamente tutti i passaggi indicati, il framework sfrutterà l'architettura già implementata per il corretto funzionamento dell'applicativo.

7.3 Altri formati di file in input

L'unico formato di data file per ora supportato, ad eccezione del file json di supporto per il repository ENCODE, è un **tab-separated values (TSV)** con sole due colonne (attributo-valore) la cui struttura è descritta nella Sezione 6.4.1. Nel caso in cui questo formato di file non fosse disponibile, è necessario definire un nuovo modulo per il popolamento della collezione di mapping Map[String, List[String]] al fine di consentire la corretta esecuzione del framework. Vista l'indipendenza delle tre sezioni del processo ETL, una volta definito il nuovo modulo estrazione, che soddisfa l'interfaccia del processo trasformazione, non è richiesta nessun'altra modifica del codice.

7.4 Cambio della tecnologia del database

Come già accennato in nella Sezione 5.2, la scelta del DBMS non influenza in alcun modo il comportamento del framework. In caso si volesse impiegare una diversa tecnologia, le operazioni da svolgere riguardano esclusivamente la modifica dei parametri di configurazione con-

tenuti nelle risorse del framework, di cui è possibile visionare un esempio nel Listing 7.1, e l'importazione della corretta libreria *driver*, implementata da Slick, nella classe **DBHandler** che deve corrispondere a quella specificata nel file di configurazione.
Una volta effettuate queste operazioni, il framework si connetterà al database utilizzando le informazioni specificate nelle risorse e sfruttando il driver. Si può notare che l'utilizzo della libreria Slick introduce questo ulteriore livello di astrazione che permette di lasciare inalterato il codice sorgente.

```
database {
   url = "jdbc:mysql://localhost:3306/geco"
   username = "root"
   password = "root"
   driver = "com.mysql.jdbc.Driver"
}
```

Listing 7.1: Esempio di configurazione dei parametri del database

7.5 Nuova tabella o attributo

L'aggiunta di una nuova tabella o attributo al modello del nostro framework comporta modifiche o, per meglio dire, aggiunte leggermente più sostanziose rispetto ai casi esaminati in precedenza.

In caso di aggiunta di una tabella o di un attributo, si ha necessità di definire i nuovi parametri all'interno della classe **DBHandler**, in modo che essi possano essere mappati al database al momento della creazione della struttura: una volta aggiunti questi parametri, si può procedere alla definizione di tutti i metodi di inserimento, modifica, ricerca nel caso si tratti di una tabella, oppure alla modifica dei metodi già esistenti nel caso si tratti di un attributo. Inoltre, per l'attributo, è necessaria un ulteriore modifica della classe a cui esso fa riferimento: questa modifica consiste nell'aggiungere il valore a livello della classe Trait <*NameTable*> e nel prevedere, in ogni implementazione di repository, la possibilità che possa essere modificato, e quindi l'adeguata gestione per ogni sorgente. Diversamente, nel caso di una nuova tabella si dovranno aggiungere sia la classe Trait che la implementa, e che svolge le funzioni di base per ogni implementazione a livello di repository, sia la classe che dovrà essere istanziata e che estenderà la classe generica per ogni repository sorgente gestito dal framework. Se la classe va a modificare alcune relazioni già presenti, ad esempio introducendo una chiave esterna in una tabella già presente, è opportuno ricordare, come descritto nella Sezione 6.6, di andare a modificare anche la classe interessata e gestire in modo corretto la relazione. Infine, si dovrà aggiungere la mappatura *valore-stringa* all'interno della classe Trait *Tables* e istanziare, in ogni classe che la estende, la classe della tabella appena definita facendo attenzione al repository di riferimento.
Slick non prevede la possibilità di aggiornare la struttura delle tabelle in modo dinamico: l'unico controllo ad ora implementato è la verifica della presenza, all'interno del database, di una tabella tramite la ricerca per nome e, nel caso questa non esista, la sua creazione utilizzando il modello descritto nel **DBHandler**. Se si fosse aggiunto un nuovo attributo alle tabelle già presenti, o ci fosse la necessità di aggiungere una nuova tabella che andrebbe a modificare la struttura

delle tabelle già presenti, sarebbe necessario procedere con la definizione della nuova struttura nel **DBHandler** per poi cancellare le tabelle presenti in modo che il framework ne ricrei la struttura prima di procedere con le operazioni di inserimento o aggiornamento. Una soluzione alternativa potrebbe essere quella di inserire manualmente la colonna modificando il DDL della tabella interessata e aggiungendo il nuovo attributo nella definizione del DBHandler: tuttavia, quest'ultima soluzione, seppur possibile, è sconsigliata in quanto potrebbe comportare errori nelle operazioni di mappatura non visibili nell'immediato.

7.6 Introduzione di una nuova statistica

Nel caso in cui si intendesse introdurre una nuova statistica all'interno del framework, sarebbe sufficiente aggiungere un nuovo contatore all'interno della classe Singleton **Statistics**, incrementandolo durante le opportune operazioni che si vogliono monitorare, ed aggiungere una nuova riga di *log*, all'interno del Main del programma, a fine esecuzione loggando la variabile e mostrando un messaggio a video facilmente interpretabile. È tuttavia consigliabile introdurre anche eventuali messaggi di log 6.9 durante l'esecuzione del framework, che monitorino l'andamento del contatore, per rendere più semplice e immediata la fase di analisi dei risultati o di debugging.

Capitolo 8

Sviluppi futuri e conclusioni

Questo lavoro ha portato alla progettazione e realizzazione di un'architettura estendibile per l'integrazione di dati (epi)genomici da sorgenti eterogenee, sfruttando un tipico processo ETL. Il framework implementa un modello comune per tutte le sorgenti che, grazie alla sua versatilità e generalità, è predisposto al supporto di future sorgenti di interesse: il supporto a nuove sorgenti non necessita di aggiunta di logica all'interno del framework ed è predisposto ad integrare singolarità della sorgente di interesse. I dati vengono scaricati e strutturati per mezzo di funzioni di trasformazione, in modo da renderli maggiormente analizzabili ed usufruibili per l'utente finale. L'indipendenza delle varie sezioni del processo ETL permette una più semplice manutenibilità del codice ed implementazione di features future. Infine, per mezzo della modalità di export dei dati, è possibile estrarre le informazioni dal database condiviso, in un formato TSV, con la finalità di utilizzo da parte di altri strumenti all'interno del progetto GeCo.
Molte sono le possibili estensioni di questo lavoro. La principale è sfruttare il pieno potenziale delle ontologie genomiche per automatizzare la gestione dei vari sinonimi che le sorgenti sul web utilizzano e per rendere le ricerche degli utenti il più indipendenti possibile dalla particolare fonte da cui proviene il dato che si sta cercando. Inoltre, le ontologie potranno essere sfruttate all'interno del framework stesso, per l'implementazione delle dipendenze di valore che ad oggi necessiterebbero di implementazioni singole per ogni sorgente di dati supportata. Oltre al normale supporto di nuove sorgenti di dati uno sviluppo futuro potrebbe concentrarsi nell'implementazione per ogni funzione di mappatura dei dati, in ingresso che necessitano di una traduzione con un numero finito di vocaboli. Tutto ciò velocizzerebbe le operazioni di aggiunta, modifica o cancellazione di regole di traduzione e allocherebbe in un unico punto tutte le funzioni di trasformazione per quanto riguarda le regole di mappatura e il vantaggio nell'automazione delle procedure compenserebbe l'inevitabile aumento dei tempi necessari al processo di trasformazione. Un applicativo di supporto, associato ad una nuova regola di mappatura, potrebbe ulteriormente aiutare a disaccoppiare il codice dalle regole di mapping.
Un'ulteriore estensione è il porting del codice verso tecnologie che utilizzando tecniche di cloud computing migliorino la parallelizzazione del processo e una corretta riprogettazione della transazionalità del codice per garantire nello stesso tempo massimo parallelismo e correttezza delle operazioni di modifica dei dati.

Bibliografia

[1] Felipe Albrecht, Markus List, Christoph Bock, and Thomas Lengauer. Deepblue epigenomic data server: programmatic data retrieval and analysis of epigenome region sets. *Nucleic acids research*, 44(W1):W581–W586, 2016.

[2] Tanya Barrett, Stephen E Wilhite, Pierre Ledoux, Carlos Evangelista, Irene F Kim, Maxim Tomashevsky, Kimberly A Marshall, Katherine H Phillippy, Patti M Sherman, Michelle Holko, et al. Ncbi geo; archive for functional genomics data sets—update. *Nucleic acids research*, 41(D1):D991–D995, 2012.

[3] Anna Bernasconi, Stefano Ceri, Alessandro Campi, and Marco Masseroli. Conceptual modeling for genomics: Building an integrated repository of open data. In *International Conference on Conceptual Modeling*, pages 325–339. Springer, 2017.

[4] Adrian Bird. Dna methylation patterns and epigenetic memory. *Genes & development*, 16(1):6–21, 2002.

[5] Adrian P Bird. Cpg-rich islands and the function of dna methylation. *Nature*, 321(6067):209–213, 1986.

[6] Erich Bornberg-Bauer and Norman W Paton. Conceptual data modelling for bioinformatics. *Briefings in bioinformatics*, 3(2):166–180, 2002.

[7] Donald F Conrad, Dalila Pinto, Richard Redon, Lars Feuk, Omer Gokcumen, Yujun Zhang, Jan Aerts, T Daniel Andrews, Chris Barnes, Peter Campbell, et al. Origins and functional impact of copy number variation in the human genome. *Nature*, 464(7289):704, 2010.

[8] 1000 Genomes Project Consortium et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061, 2010.

[9] ENCODE Project Consortium et al. An integrated encyclopedia of dna elements in the human genome. *Nature*, 489(7414):57, 2012.

[10] Fabio Cumbo, Giulia Fiscon, Stefano Ceri, Marco Masseroli, and Emanuel Weitschek. Tcga2bed: extracting, extending, integrating, and querying the cancer genome atlas. *BMC bioinformatics*, 18(1):6, 2017.

[11] Encode. Encyclopedia of dna element. `https://www.encodeproject.org/`.

[12] Javier D Fernández, Maurizio Lenzerini, Marco Masseroli, Francesco Venco, and Stefano Ceri. Ontology-based search of genomic metadata. *IEEE/ACM transactions on computational biology and bioinformatics*, 13(2):233–247, 2016.

[13] GEO. The gene expression omnibus. `https://www.ncbi.nlm.nih.gov/geo/`.

[14] Joachim Hammer and Markus Schneider. Genomics algebra: A new, integrating data model, language, and tool for processing and querying genomic information. In *CIDR*, volume 2, pages 176–187. Citeseer, 2003.

[15] Erika Check Hayden. Is the 1,000 genome for real? *Nature News*, 2014.

[16] Thomas Hernandez and Subbarao Kambhampati. Integration of biological sources: current systems and challenges ahead. *ACM SIgmod Record*, 33(3):51–60, 2004.

[17] Vahid Jalili, Matteo Matteucci, Marco Masseroli, and Stefano Ceri. Indexing next-generation sequencing data. *Information Sciences*, 384:90–109, 2017.

[18] Anshul Kundaje, Wouter Meuleman, Jason Ernst, Misha Bilenky, Angela Yen, Alireza Heravi-Moussavi, Pouya Kheradpour, Zhizhuo Zhang, Jianrong Wang, Michael J Ziller, et al. Integrative analysis of 111 reference human epigenomes. *Nature*, 518(7539):317, 2015.

[19] Bo Li and Colin N Dewey. Rsem: accurate transcript quantification from rna-seq data with or without a reference genome. *BMC bioinformatics*, 12(1):323, 2011.

[20] Marco Masseroli, Abdulrahman Kaitoua, Pietro Pinoli, and Stefano Ceri. Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying. *Methods*, 111:3–11, 2016.

[21] Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature methods*, 5(7):621, 2008.

[22] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[23] Genomes Project. 1000 genomes project. `http://www.internationalgenome.org/`.

[24] RoadMap. Road map epigenomimc projects. `http://www.roadmapepigenomics.org/`.

[25] Abhishek Roy, Yanlei Diao, Uday Evani, Avinash Abhyankar, Clinton Howarth, Rémi Le Priol, and Toby Bloom. Massively parallel processing of whole genome sequence data: an in-depth performance study. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 187–202. ACM, 2017.

[26] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135, 2008.

[27] Slick. Functional relational mapping for scala. http://slick.lightbend.com/.

[28] TCGA. The cancer genome atlas. https://gdc.cancer.gov/.

[29] Jorge Ignacio Vera Pena. Automation of retrieval, transformation and uploading of genomic data and their metadata for their integration into a gdm repository. 2017.

[30] John N Weinstein, Eric A Collisson, Gordon B Mills, Kenna R Mills Shaw, Brad A Ozenberger, Kyle Ellrott, Ilya Shmulevich, Chris Sander, Joshua M Stuart, Cancer Genome Atlas Research Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature genetics*, 45(10):1113, 2013.

[31] Emanuel Weitschek, Daniele Santoni, Giulia Fiscon, Maria Cristina De Cola, Paola Bertolazzi, and Giovanni Felici. Next generation sequencing reads comparison with an alignment-free distance. *BMC research notes*, 7(1):869, 2014.

[32] Yan Zeng and Bryan R Cullen. Sequence requirements for micro rna processing and function in human cells. *Rna*, 9(1):112–123, 2003.

Appendice A

Tabelle di mapping per il repository Encode

In questa sezione vengono presentate tutte le regole di mapping utilizzate per importare i dati della sorgente ENCODE al repository condiviso.
La colonna *Attributo* indica a quale colonna della tabella il mapping fa riferimento. La colonna *SourceKey* indica il nome dell'attributo (o chiave) del file in input al programma contenente tutte le informazioni che si vogliono mappare nella colonna *Attributo*, mentre *Metodo* indica il metodo utilizzato per mappare la *SourceKey* nell'attributo. Nel caso in cui siano presenti più *SourceKey* per un singolo attributo i metodi di mapping sono da considerarsi applicati usando la *SourceKey* nella stessa riga della tabella e con ordine di lettura (dall'alto verso il basso).
Con il termine <prefix> nella tabella si vuole indicare la stringa *replicates__X__library__biosample__donor*: la X specifica che in quella posizione può esserci un qualsiasi numero, infatti per ENCODE è possibile che in ogni file siano presenti riferimenti a più donatori, campioni biologici o replicati.

Donor		
Attributo	**Encode**	
	SourceKey	Metodo
SourceId	<prefix>__accession	Default
Species	<prefix>__organism__scientific_name	Default
Age	<prefix>__age_display	Default
Gender	<prefix>__sex	Default
Ethnicity	<prefix>__ethnicity	Default

Tabella A.1: Mapping tabella Donor per repository ENCODE

BioSample		
Attributo	**Encode**	
	SourceKey	Metodo
SourceId	<prefix>__accession	Default
Types	<prefix>__biosample_type	Default
Tissue	<prefix>__biosample_term_id; <prefix>__biosample_term_name	Default; Concat
CellLine	<prefix>__biosample_term_id; <prefix>__biosample_term_name	Default; Concat
isHealty	<prefix>__health_status	Default
Disease	<prefix>__health_status	Default

Tabella A.2: Mapping tabella BioSample per repository ENCODE

Replicate		
Attributo	**Encode**	
	SourceKey	Metodo
SourceId	<prefix>__uuid	Default
BioReplicateNum	<prefix>__biological_replicate_number	Default
TechReplicateNum	<prefix>__technical_replicate_number	Default

Tabella A.3: Mapping tabella Replicate per repository ENCODE

Item		
Attributo	**Encode**	
	SourceKey	Metodo
SourceId	file__accession	Default
DataType	file__output_type	Default
Format	file__file_type	Default
Size	file__file_size	Default
Platform	routine di ricerca	-
Pipeline	routine di ricerca	-
SourceUrl	https://www.encodeproject.org; file__href	Manually; Concat
LocalUrl	-	-

Tabella A.4: Mapping della tabella Item per repository ENCODE

Container		
Attributo	Encode	
	SourceKey	Metodo
Name	file__assembly; award__project; file__file_format_type	Sub-Concat; Sub-Concat; Sub-Remove-Concat
Assembly	file__assembly	Default
isAnn	false	Manually
Annotation	null	Manually

Tabella A.5: Mapping della tabella Container per repository ENCODE

ExperimentType		
Attributo	ENCODE	
	SourceKey	Metodo
Technique	assay_term_name	Default
Feature	file__replicate__experiment__target__investigated_as; target__investigated_as	Default; CheckPrec
Target	target__label	Concat
Antibody	replicates__X__antibody__title; replicates__X__antibody__url	Default; Concat

Tabella A.6: Mapping della tabella ExperimentType per repository ENCODE

Case		
Attributo	Encode	
	SourceKey	Metodo
SourceId	accession	Default
SourceSite	submitted_by__lab	Remove-Sub
ExternalRef	dbxrefs	Default

Tabella A.7: Mapping della tabella Case per repository ENCODE

Project		
Attributo	Encode	
	SourceKey	Metodo
ProjectName	ENCODE	Manually
ProgramName	award__project	Default

Tabella A.8: Mapping della tabella Project per repository ENCODE

Appendice B

Tabelle di mapping per il repository TCGA

In questa sezione vengono presentate tutte le regole di mapping utilizzate per importare i dati della sorgente TCGA al repository condiviso.
La colonna *Attributo* indica a quale colonna della tabella il mapping fa riferimento. La colonna *SourceKey* indica il nome dell'attributo (o chiave) del file in input al programma contenente tutte le informazioni che si vogliono mappare nella colonna *Attributo*, mentre *Metodo* indica il metodo utilizzato per mappare la *SourceKey* nell'attributo. Nel caso in cui siano presenti più *SourceKey* per un singolo attributo i metodi di mapping sono da considerarsi applicati usando la *SourceKey* nella stessa riga della tabella e con ordine di lettura (dall'alto verso il basso).

Donor		
Attributo	**TCGA**	
	SourceKey	Metodo
SourceId	clinical__shared__bcr_patient_uuid; biospecimen__shared__bcr_patient_uuid	Deafult; CheckPrec
Species	Homo sapiens	Manually
Age	manually_curated__cases__demographic__year_of_birth	Deafult
Gender	clinical__shared__gender; biospecimen__shared__gender	CheckPrec
Ethnicity	clinical__clin_shared__race; clinical__clin_shared__ethnicity	Concat

Tabella B.1: Mapping della tabella Donor per repository TCGA

BioSample		
Attributo	TCGA	
	SourceKey	Metodo
SourceId	biospecimen__bio__bcr_sample_uuid	Default
Types	tissue	Manually
Tissue	manually_curated__cases__primary_site	Default
CellLine	null	Manually
isHealty	biospecimen__bio__sample_type	Default
Disease	manually_curated__cases__disease_type	Default

Tabella B.2: Mapping della tabella Biosample per repository TCGA

Replicate		
Attributo	TCGA	
	SourceKey	Metodo
SourceId	biospecimen__bio__bcr_sample_uuid	Default
BioReplicateNum	1	Manually
TechReplicateNum	1	Manually

Tabella B.3: Mapping della tabella Biosample per repository TCGA

Item		
Attributo	TCGA	
	SourceKey	Metodo
SourceId	manually_curated__file_id	Default
DataType	manually_curated__data_type	Default
Format	manually_curated__source_data_format	Default
Size	manually_curated__file_size	Default
Platform	manually_curated__platform	Deafult
Pipeline	manually_curated__analysis__workflow_type	Default
SourceUrl	https://portal.gdc.cancer.gov/files/; manually_curated__file_id	Manually, Concat
LocalUrl	url locale	Default

Tabella B.4: Mapping della tabella Item per il repository TCGA

Container		
Attributo	TCGA	
	SourceKey	Metodo
Name	GRCh38; manually_curated__cases__project__program__name; manually_curated__data_type	Manually; Sub-Concat; Sub-Concat
Assembly	GRCh38	Default
isAnn	false	Manually
Annotation	null	Manually

Tabella B.5: Mapping della tabella Container per repository TCGA

ExperimentType		
Attributo	TCGA	
	SourceKey	Metodo
Technique	manually_curated__experimental_strategy	Default
Feature	manually_curated__experimental_strategy	Default
Target	null	Manually
Antibody	null	Manually

Tabella B.6: Mapping della tabella ExperimentType per repository TCGA

Case		
Attributo	TCGA	
	SourceKey	Metodo
SourceId	manually_curated__cases__case_id	Concat
SourceSite	clinical__shared__tissue_source_site	Default
ExternalRef	Null	Manually

Tabella B.7: Mapping della tabella Case per repository TCGA

Project		
Attributo	TCGA	
	SourceKey	Metodo
ProjectName	manually_curated__cases__project__program__name; clinical__admin__disease_code	Deafult; Concat
ProgramName	manually_curated__cases__project__program__name	Default

Tabella B.8: Mapping della tabella Project per repository TCGA

Appendice C

Schema xsm del file di configurazione

```
1  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
       xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="root">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="table" maxOccurs="unbounded" minOccurs="0">
6            <xs:complexType>
7              <xs:sequence>
8                <xs:element name="mapping" maxOccurs="unbounded"
       minOccurs="0">
9                  <xs:complexType>
10                   <xs:sequence>
11                     <xs:element type="xs:string" name="source_key"
       />
12                     <xs:element type="xs:string" name="global_key"
                           />
13                   </xs:sequence>
14                   <xs:attribute type="xs:string" name="method" use="
                       optional"/>
15                   <xs:attribute type="xs:string" name="
                       concat_character" use="optional"/>
16                   <xs:attribute type="xs:string" name="sub_character
                       " use="optional"/>
17                   <xs:attribute type="xs:string" name="new_character
                       " use="optional"/>
18                   <xs:attribute type="xs:string" name="rem_character
                       " use="optional"/>
19                 </xs:complexType>
20               </xs:element>
21             </xs:sequence>
22             <xs:attribute type="xs:string" name="name"/>
23           </xs:complexType>
24         </xs:element>
25       </xs:sequence>
26     </xs:complexType>
27   </xs:element>
```

```
28 </xs:schema>
```

Listing C.1: Schema xsm utilizzato per validare il file di configurazione in input

Appendice D

Confronto proprietà attributi modello ideale e modello reale

In questa sezione sono elencate, per favorirne il cofronto, tutte le proprietà degli attributi delle tabelle sia nel modello ideale che in quello reale. È inoltre espresso il tipo del dato all'interno del framework.

Item			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
SourceId	[SM]	[SM]	String
DataType	[RSM]	[RS]	String
Format	[D(*DataType*)RSM]	[RS]	String
Size	[SM]	[ST]	Int
SourceUrl	[M]	[S]	String
LocalUrl	[C(*Format*)SM]	[S]	String
Pipeline	[D(*Technique*)S]	[C(*DataType*)S]	String
Platform	[C(*DataType*)RSM]	[C(*DataType*)S]	String

Tabella D.1: Proprietà attributi entità Item

Donor			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
SourceId	[SM]	[SM]	String
Species	[RSM]	[RS]	String
Age	[S]	[ST]	String
Gender	[RS]	[S]	String
Ethinicity	[RS]	[S]	String

Tabella D.2: Proprietà attributi entità Donor

Biosample			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
SourceId	[M]	[SM]	String
Type	[RSM]	[S]	String
Tissue	[CSMO]	[C(*Type*)S]	String
CellLine	[CSM O]	[C(*Type*)S]	String
IsHealthy	[RS]	[RS]	Boolean
Disease	[C(*IsHealthy*)D(*Tissue*)O]	[S]	String

Tabella D.3: Proprietà attributi entità Biosample

Replicate			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
SourceId	[SM]	[SM]	String
bioReplicateNum	[S]	[S]	Int
techReplicateNum	[S]	[S]	Int

Tabella D.4: Proprietà attributi entità Replicate

ExperimentType			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
Technique	[RSM]	[SM]	String
Feature	[D(*Technique*)RSMH]	[S]	String
Platform	[C(*Datatype*)RSM]	-	-
Target	[C(*Technique*)RSM]	[S]	String
Antibody	[C(*Technique*)D(*Target*)RSM]	[S]	String

Tabella D.5: Proprietà attributi entità ExperimentType

Container			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
Name	[SM]	[SM]	String
Assembly	[C(*Datatype*)D(*Species*)RSM]	[D(*Species*)S]	String
isAnn	[RSM]	[RS]	Boolean
Annotation	[C(*IsAnn*)RSM]	[C(*isAnn*)S]	String

Tabella D.6: Proprietà attributi entità Container

Case			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
SourceId	[SM]	[SM]	String
SourceSite	[SM]	[S]	String
ExternalRef	[]	[S]	String

Tabella D.7: Proprietà attributi entità Case

Project			
Attributo	Modello Ideale	Modello Reale	Tipo Dato
ProjectName	[SM]	[SM]	String
ProgramName	[S]	[S]	String

Tabella D.8: Proprietà attributi entità Project

Appendice E

Statistiche delle esecuzioni

In questa sezione vengono mostrate, in un formato più dettagliato, le statistiche collezionate dalle esecuzioni del framework sui diversi dataset. Vengono elencate media, deviazione standard e valore dell'intervallo di confidenza al 90%.

Statistica	derived_from	support_table_insert	File Inserted	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	VERO	VERO	10534	16884	56370	73254	0.01.48	0.00.55	2.28.55	2.31.37	8,06
Dev Std							0.00.52	0.00.03	0.03.44	0.04.28	0,24
Confidenza							0.00.49	0.00.03	0.03.32	0.04.15	0,22

Tabella E.1: Statistiche per il dataset HG19 di ENCODE con 3 campioni.

Statistica	derived_from	support_table_insert	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	FALSO	FALSO	10534	0	10534	0.00.51	0.01.03	0.38.17	0.40.11	4,37
Dev Std						0.00.11	0.00.15	0.01.21	0.00.56	0,10
Confidenza						0.00.10	0.00.14	0.01.17	0.00.53	0,10

Tabella E.2: Statistiche per il dataset HG19 di ENCODE con 3 campioni.

Statistica	derived_from	support_table_insert	File Inserted	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	VERO	VERO	10091	17307	56275	73582	0.01.45	0.00.46	2.09.40	2.12.12	9,33
Dev Std							0.00.58	0.00.03	0.09.27	0.10.15	0,74
Confidenza							0.00.56	0.00.03	0.08.58	0.09.44	0,70

Tabella E.3: Statistiche per il dataset GRCh38 di ENCODE con 3 campioni.

Statistica	derived_from	support_table_insert	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	FALSO	FALSO	10091	0	10091	0.01.10	0.00.46	0.39.45	0.41.41	4,04
Dev Std						0.00.09	0.00.02	0.01.58	0.01.53	0,18
Confidenza						0.00.09	0.00.02	0.01.52	0.01.47	0,17

Tabella E.4: Statistiche per il dataset GRCh38 di ENCODE con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	22374	0	22374	0.02.13	0.03.02	0.58.10	1.03.25	5,83
Dev Std				0.01.39	0.00.03	0.01.09	0.01.39	0,13
Confidenza				0.01.34	0.00.03	0.01.06	0.01.34	0,13

Tabella E.5: Statistiche per il dataset copy_number_segment di TCGA con 4 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	10999	0	10999	0.01.35	0.01.35	0.32.16	0.35.26	5,18
Dev Std				0.01.26	0.00.02	0.00.39	0.01.48	0,26
Confidenza				0.01.21	0.00.01	0.00.37	0.01.42	0,25

Tabella E.6: Statistiche per il dataset isoform_expression_quantification di TCGA con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	10947	0	10947	0.00.48	0.01.32	0.31.02	0.33.23	5,47
Dev Std				0.00.08	0.00.03	0.00.44	0.00.54	0,15
Confidenza				0.00.07	0.00.03	0.00.42	0.00.51	0,14

Tabella E.7: Statistiche per il dataset mirna_expression_quantification di TCGA con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	11091	0	11091	0.01.36	0.01.32	0.31.23	0.34.31	5,36
Dev Std				0.01.28	0.00.00	0.00.37	0.00.52	0,13
Confidenza				0.01.23	-	0.00.36	0.00.49	0,13

Tabella E.8: Statistiche per il dataset gene_expression_quantification di TCGA con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	22375	0	22375	0.03.13	0.03.23	1.10.26	1.17.02	4,95
Dev Std				0.02.59	0.00.42	0.15.21	0.14.49	0,87
Confidenza				0.02.50	0.00.40	0.14.34	0.14.04	0,83

Tabella E.9: Statistiche per il dataset masked_copy_number_segment di TCGA con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
MEDIA	10187	0	10187	0.01.40	0.01.29	0.29.22	0.32.31	5,22
Dev STD				0.01.17	0.00.03	0.01.08	0.00.10	0,03
Confidenza				0.01.13	0.00.03	0.01.04	0.00.09	0,03

Tabella E.10: Statistiche per il dataset masked_somatic_mutation di TCGA con 3 campioni.

Statistica	Item Inserted	Item Updated	Total Item	extract	Transform	Load	Total	Item/sec
Media	12218	0	12218	0.01.09	0.01.42	0.35.25	0.38.16	5,33
Dev Std				0.00.41	0.00.04	0.01.54	0.01.39	0,22
Confidenza				0.00.39	0.00.04	0.01.48	0.01.34	0,21

Tabella E.11: Statistiche per il dataset masked_copy_number_segment di TCGA con 3 campioni.