ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche LM

Progettazione di Data Warehouse di dati genomici su piattaforma Hadoop

Tesi in
DATA MINING

Relatore
Prof. MATTEO GOLFARELLI

Co-relatore
Dott. SIMONE GRAZIANI

Presentata da
DRUDI RICCARDO

Terza sessione di Laurea
Anno Accademico 2014 - 2015

Alla mia famiglia, a Sofia, ai Good Ole Boys

Sommario

Introduzione 8
1 Le piattaforme Apache Hadoop e Apache Spark 10
1.1 Apache Hadoop 10
1.1.1 Hadoop Distributed File System (HDFS) 11
1.1.2 MapReduce 13
1.1.3 Hadoop Yarn 14
1.2 Apache Spark 17
1.2.1 Componenti 19
1.2.2 Architettura 21
1.2.3 Scheduling 22
1.2.4 RDD Abstraction 24
1.2.5 Componenti di esecuzione: Job, Task e Stage 26
1.2.6 DataFrame 28
2 Analisi di dati genomici 30
2.1 Obiettivi e problematiche 30
2.2 Stato dell'arte 32
2.3 Il database TCGA 34
2.3.1 Portale Dati 34
2.3.2 Data Level 35
2.3.3 Data Type 36
2.3.4 Come ottenere i Dati 38
2.3.5 Mole di dati 38
2.4 Il linguaggio GMQL e il Genome Space 39
2.4.1 Genometric Query Language (GMQL) 40
2.4.2 Genome Space 41
3 Scoperta automatica di dipendenze funzionali approssimate 42
3.1 Definizione del problema 42
3.1.1 Dipendenze funzionali esatte 42

3.1.2 Dipendenze funzionali approssimate 43
3.1.1 Dipendenze funzionali condizionate 43
3.2 Stato dell'arte 44
3.3 Algoritmi 45
3.3.1 TANE 45
3.3.2 FUN 48
3.3.3 FD_mine 50
3.4 Implementazione distribuita su Apache Spark 51
3.4.1 Calcolo di Dipendenze Funzionali Approssimate (AFDs) 53
4 Modellazione multidimensionale del Genome Space 57
4.1 Scoperta di gerarchie dal database TCGA 57
4.1.1 Metodologia 57
4.1.2 Risultati 61
4.2 Il cubo Mapping 65
5 Conclusioni 73
6 Bibliografia 75

Introduzione

La bioinformatica è una disciplina che unisce analisi di carattere biologico con l'utilizzo dell'informatica come mezzo di supporto. Negli ultimi anni la biologia ha fatto ricorso in misura sempre maggiore all'informatica per affrontare analisi complesse che prevedono l'utilizzo di grandi quantità di dati e tali analisi costituiscono una delle nuove sfide tecnologiche dell'informatica. Fra le scienze biologiche che prevedono l'elaborazione di grandi quantità di dati c'è la genomica, una branca della biologia molecolare che si occupa dello studio di struttura, contenuto, funzione ed evoluzione del genoma degli organismi viventi. Lo sviluppo delle conoscenze genetiche in medicina e nella pratica clinica consente di eseguire studi basati sul sequenziamento del genoma umano e mirati a determinare l'incidenza di una specifica patologia su un campione o su un individuo rispetto alla popolazione generale. I sistemi di data warehouse sono una tecnologia informatica che ben si adatta a supportare determinati tipi di analisi in ambito genomico perché consentono di effettuare analisi esplorative e dinamiche, analisi che si rivelano utili quando si vogliono ricavare informazioni di sintesi a partire da una grande quantità di dati e quando si vogliono esplorare prospettive e livelli di dettaglio diversi.

Il lavoro di tesi si colloca all'interno di un progetto più ampio riguardante la progettazione di un data warehouse in ambito genomico a partire da una sorgente dati di 11TB comprendente informazioni cliniche, immagini e diversi tipi di analisi sul sequenziamento genomico. I dati utilizzati in questo lavoro di tesi sono dati forniti dal National Cancer Institute (NCI) che fa parte del National Institute of Health (NIC) all'interno del Dipartimento della Salute e dei Servizi Umani degli Stati Uniti. L'NCI, attraverso diversi investimenti, ha supportato la creazione del progetto TCGA (Cancer Genome Atlas). Il TCGA ha lo scopo di creare un catalogo delle mutazioni genetiche responsabili del cancro, usando sequenze genomiche e bioinformatica. I dati specifici su cui ci si è concentrati nel lavoro di tesi riguardano principalmente i dati clinici dei pazienti, cioè i dati relativi al sesso, all'etnia, all'età, allo stato della malattia etc. Le analisi effettuate hanno portato alla scoperta di dipendenze funzionali e di conseguenza alla definizione di una gerarchia nei dati. Attraverso l'inserimento di tale gerarchia in un modello

multidimensionale relativo ai dati genomici sarà possibile ampliare il raggio delle analisi da poter eseguire sul data warehouse introducendo un contenuto informativo ulteriore riguardante le caratteristiche dei pazienti.

I passi effettuati in questo lavoro di tesi sono stati prima di tutto il caricamento e filtraggio dei dati per eliminare attributi con un numero di istanze poco significativo. Il fulcro del lavoro di tesi è stata l'implementazione di un algoritmo per la scoperta di dipendenze funzionali con lo scopo di ricavare dai dati una gerarchia. Nell'ultima fase del lavoro di tesi si è inserita la gerarchia ricavata all'interno di un modello multidimensionale preesistente in modo da poter ampliare il progetto di un data warehouse di dati genomici.

L'intero lavoro di tesi è stato svolto attraverso l'utilizzo di Apache Spark e Apache Hadoop, un framework open-source per la memorizzazione e la gestione distribuita di grandi data set su cluster di computer.

1 Le piattaforme Apache Hadoop e Apache Spark

1.1 Apache Hadoop

Apache Hadoop è un framework open-source scritto in Java per la memorizzazione e la gestione distribuita di grandi data set su cluster di computer costruiti con commodity hardware. Tutti i moduli presenti in Hadoop sono progettati con la fondamentale assunzione che i "guasti hardware" sono comuni e che devono essere automaticamente gestiti dal software del framework.

Il "core" di hadoop consiste in una parte di memorizzazione (**Hadoop Distribute File System HDFS**) e in una parte di elaborazione (**MapReduce**). Hadoop divide i file in grandi blocchi e li distribuisce lungo i nodi del cluster. Per processare i dati, Hadoop MapReduce trasferisce il codice nei nodi per poter processare in parallelo, sulla base dei dati che ogni nodo deve elaborare.

Il framework base di Hadoop, come si evince dalla Figura 1, si compone dei seguenti moduli:

- *Hadoop Common*: contiene librerie e utility utilizzate dagli altri moduli Hadoop
- *Hadoop Distributed File System (HDFS)*: un file-System distribuito che memorizza dati su macchine, offrendo un'elevata larghezza di banda all'interno del cluster
- *Hadoop YARN*: una piattaforma di gestione delle risorse responsabile della gestione delle risorse nel cluster e utilizzata per la schedulazione delle applicazioni utente.
- *Hadoop MapReduce*: un modello di programmazione per l'elaborazione di grandi moli di dati.

Il termine Hadoop però non si riferisce soltanto ai moduli base, ma a tutto l'ecosistema [1], quindi all'insieme di software aggiuntivi che possono essere installati come *Apache Pig*, *Apache Hive*, *Apache Hbase*, ***Apache Spark*** e altri [2].I componenti HDFS e MapReduce sono stati ispirati dalle pubblicazioni di google su *MapReduce* e *Google File System*.

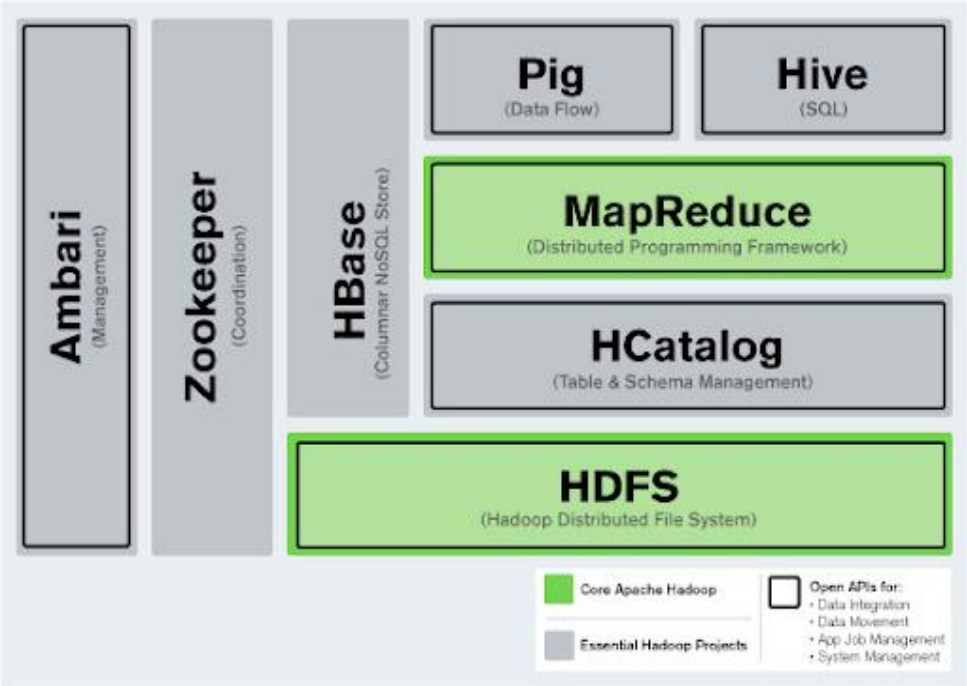


Figura 1 Framework Hadoop: in figura sono evidenziati gli elementi core che compongono Apache Hadoop

1.1.1 Hadoop Distributed File System (HDFS)

L'Hadoop Distributed File System (HDFS) è un file System distribuito, scalabile e portabile scritto in Java per il framework Hadoop. Un cluster Hadoop è nominalmente un singolo NameNode più un gruppo di DataNode, anche se è possibile un'opzione di ridondanza anche per il namenode data la sua criticità. Ogni datanode serve blocchi di dati sulla rete usando un protocollo specifico per HDFS. Il file-system utilizza le socket TCP/IP per la comunicazione. I client invece utilizzano Remote Procedure Call (RPC) per comunicare tra loro.

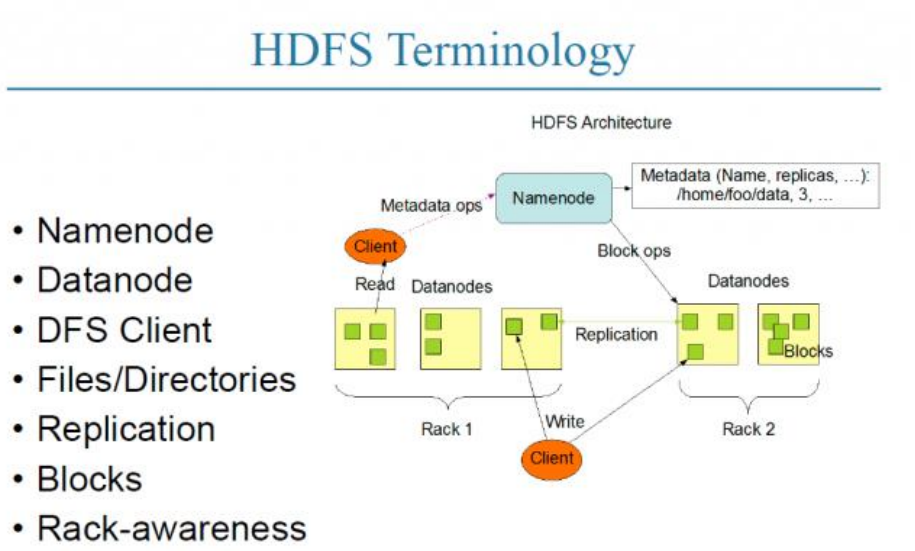


Figura 2 Architettura di HDFS

HDFS memorizza grandi file (generalmente gigabyte o terabyte) su più macchine. Raggiunge l'affidabilità del dato replicando i dati su più host. Con il valore di replicazione di default (3) il dato è salvato su 3 nodi, di cui 2 sullo stesso rack e uno su un rack differente (vedi Figura 2). I datanode possono comunicare tra loro per ribilanciare i dati, per spostare copie, e per mantenere un'elevata replica dei dati.

Il File System HDFS include anche un namenode secondario. Il namenode secondario si connette regolarmente con il primario e costruisce una snapshot delle directory information del namenode primario che poi il sistema salva in directory locali o remote. Queste immagini di checkpoint possono essere usate per fare un restart del namenode primario senza dover replicare tutte le action del file system. Dato che il namenode è un punto singolo per immagazzinare e gestire i metadati, può diventare un collo di bottiglia (bottleneck) per supportare un gran numero di file, specialmente un gran numero di piccoli file. HDFS federation, una nuova aggiunta, si propone di affrontare questo problema in una certa misura, consentendo molteplici name-space serviti da namenode separati.

Un vantaggio nell'uso di HDFS è la consapevolezza dei dati tra il job tracker e il task tracker. Il job tracker schedula operazioni di map o reduce al task tracker con la consapevolezza della localizzazione dei dati.

In questo modo si riduce la quantità di traffico lungo la rete e si prevengono trasferimenti di dati non necessari. Questo potrebbe avere un impatto significativo sul tempo di completamento dei lavori. Quando Hadoop viene utilizzato con altri file system questo vantaggio non sempre è disponibile. Questo potrebbe avere un impatto significativo sul tempo di completamento dei job.

1.1.2 MapReduce

Sopra il File System si posiziona il motore MapReduce, che consiste in un JobTracker a cui le applicazioni client inviano i lavori MapReduce. Il JobTracker direziona i lavori nei nodi TaskTraker disponibili nel cluster, cercando di mantenere il lavoro più vicino possibile ai dati.

Con un File System Rack-aware, il JobTracker conosce quali nodi contengono i dati e quali macchine si trovano nelle vicinanze. Se il lavoro non può essere ospitato sul nodo effettivo in cui risiedono i dati, viene data priorità ai nodi dello stesso rack. Questo riduce il traffico di rete sulla rete backbone principale.

Se un TaskTracker fallisce o va in timeout, quella parte del job viene rischedulata. Il TaskTracker su ogni nodo crea un processo della Java Virtual Machine separato per prevenire il fail del TaskTracker stesso se il job corrente blocca la JVM. Un segnale viene mandato dal TaskTracker al JoTracker ogni tanto (qualche minuto) per controllare il suo stato. Gli stati del JobTracker e del TaskTracker e le altre informazioni sono esposte da Jetty e visualizzabili da un web browser.

Oltre al JobTracker un programma MapReduce si compone anche di un Mapper e di un Reducer.

Il codice del **mapper** legge i file di input come coppie chiave-valore (<*Key*,*Value*>) e restituisce sempre coppie chiave valore. La classe Mapper estende *MapReduceBase* e implementa l'interfaccia *Mapper*. L'interaccia mapper si aspetta quattro generici, che definiscono i tipi delle coppie chiave-valore di input e di output. I primi due parametri definiscono il tipo della chiave e il tipo del valore di input, i secondi due definiscono il tipo della chiave e il tipo del valore di output. La funzione map() accetta come input la chiave, il valore, l'*OutputCollector* per i

valori in output e un oggetto *Reporter*. L'OutputCollector è responsabile della scrittura dei dati intermedi generati dal mapper.

Il codice del **reducer** legge gli output generati dai differenti mapper come coppie chiave-valore (<*Key*,*Value*>) e restituisce coppie chiave-valore. La classe Reducer estende *MapReduceBase* e implementa l'interfaccia *Reducer*. L'interfaccia Reducer si aspetta 4 generici, che definiscono i tipi delle coppie chiave-valore di input e output. I primi due parametri definiscono i tipi della chiave e del valore intermedi, i secondi due parametri definiscono i tipi della chiave e del valore finali. Le chiavi sono *WritableComparables* mentre i valori sono *Writables*.

La funzione reduce() accetta come input la chiave, un iteratore, un *OutputCollector* e un oggetto *Reporter*. L'OutputCollector è responsabile della scrittura del risultato finale in output [3].

1.1.3 Hadoop Yarn

MapReduce ha subito una revisione completa in Hadoop 0.23 e in questo momento è disponibile MapReduce 2.0 (MRv2) detto YARN.

Apache Hadoop YARN è un sotto-progetto di Hadoop e, come si può vedere dalla Figura 4, è stato introdotto in Hadoop 2.0 con il compito di separare la gestione delle risorse dai componenti di processamento. YARN è nato dalla necessità di consentire una più ampia gamma di modelli di interazione per i dati memorizzati in HDFS oltre MapReduce. L'architettura YARN-based di Hadoop 2.0 (vedi Figura 3) fornisce una piattaforma di calcolo più generale che non è vincolata a MapReduce.

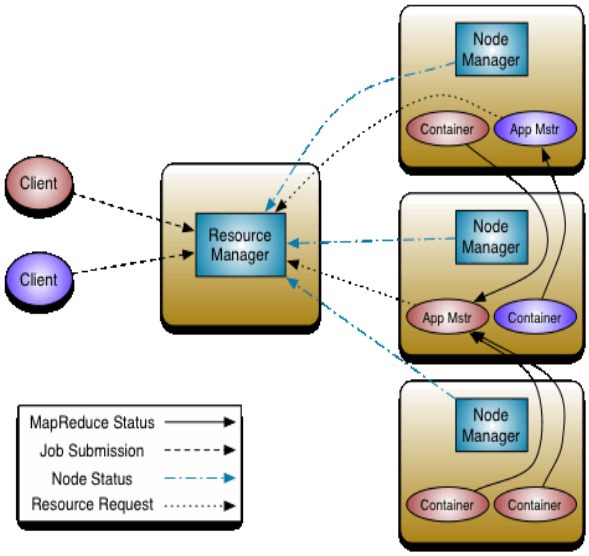


Figura 3 Architettura di Hadoop YARN

L'idea fondamentale di MRv2 è quella di dividere le due principali funzioni del JobTracker, cioè la gestione delle risorse (resource management) e la schedulazione e il monitoraggio dei lavori (job scheduling/monitoring), in due demoni separati. Si vuole quindi avere un ResourceManager (RM) globale e un ApplicationManager (AM) per applicazione.

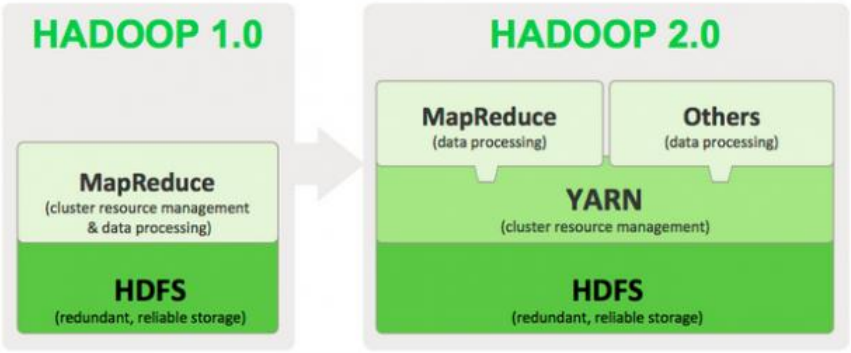


Figura 4 Evoluzione del Framework Haddop da Hadoop 1.0 a Hadoop 2.0; si può notare l'introduzione di YARN

Come parte di Hadoop 2.0, YARN usa le capacità di gestione delle risorse che erano in MapReduce e le confeziona in modo che possano essere utilizzate da nuovi motori. Questo semplifica e limita il lavoro di MapReduce, infatti con YARN MapReduce dovrà solo preoccuparsi di processare i dati. YARN è inoltre possibile l'esecuzione di più

applicazioni Hadoop, tutte accomunate da una gestione comune delle risorse.

YARN aumenta la potenza di calcolo dei cluster Hadoop nei seguenti modi:

- *Scalabilità*: la potenza di elaborazione nei data center continua a crescere rapidamente. Dato che il ResourceManager YARN si concentra esclusivamente sulla pianificazione, può gestire cluster di grandi dimensioni molto più facilmente.
- *Compatibilità con MapReduce*: le applicazioni MapReduce esistenti e gli utenti possono eseguire su YARN senza dover riscrivere i processi esistenti in quanto le applicazioni scritte con le vecchie API continueranno a funzionare.
- *Miglior utilizzo del Cluster*: il ResourceManager è uno scheduler puro che ottimizza l'utilizzo del cluster in base ai criteri quali garanzie di capacità, equità, e Service Level Agreement.
- *Supporto per carico di lavoro diverso da MapReduce*: modelli di programmazione aggiuntivi come il graph processing e l'iterative modeling sono ora possibili per il processamento dei dati. Questi modelli aggiuntivi consentono alle aziende di realizzare dei processi real-time.
- *Agilità*: può evolvere indipendentemente dal layer di gestione di risorse sottostante in una maniera più agile di MapReduce.

L'idea fondamentale di YARN è di dividere le due responsabilità maggiori di JobTracker/TaskTracker in entità separate:

- Un ResourceManager globale
- Un ApplicationMaster per applicazione
- Un NodeManager per nodo
- Un container per applicazione eseguito su un NodeManager

Il ResourceManager e il NodeManager formano un nuovo, e generico, sistema di gestione delle applicazioni in modo distribuito. Il ResourceManager è l'autorità suprema che arbitra risorse tra tutte le applicazioni presenti nel sistema. L'ApplicationMaster è un entità framework-specific e ha il comptio di negoziare risorse dal

ResourceManager e lavorare con i NodeManager per eseguire e monitorare le attività dei componenti. Il ResourceManager ha uno scheduler, che è responsabile per l'assegnazione delle risorse alle varie applicazioni in esecuzione, nel rispetto dei vincoli, quali le capacità di coda, user-limit etc. lo scheduler svolge la sua funzione di schedula sulla base dei requisiti delle risorse delle applicazioni. Il NodeManager è lo slave per macchina, che è responsabile del lancio dell'application container, del controllo dell'utilizzo delle risorse (CPU, memoria, disco, rete) e di fare il report dello stesso al ResourceManager. Ogni ApplicationMaster ha la responsabilità di negoziare appositi contenitori di risorse dallo scheduler, monitorare lo stato e monitorare i progressi. Dal punto di vista del sistema, l'ApplicationMaster viene eseguito come un normale container [4] [5] [6].

1.2 Apache Spark

Apache Spark è un framework open source cluster originariamente sviluppato nel AMPLab dell'università della California, Berkeley ma è stato successivamente donato all'*Apache Software Foundation*. A differenza del paradigma MapReduce di Hadoop, le primitive di spark forniscono performance fino a 100 volte più veloci per determinate applicazioni [7]. Consentendo ai programmi utente di caricare i dati nella memoria cluster e interrogarli ripetutamente. Spark è anche adatto per algoritmi machine learning [8].

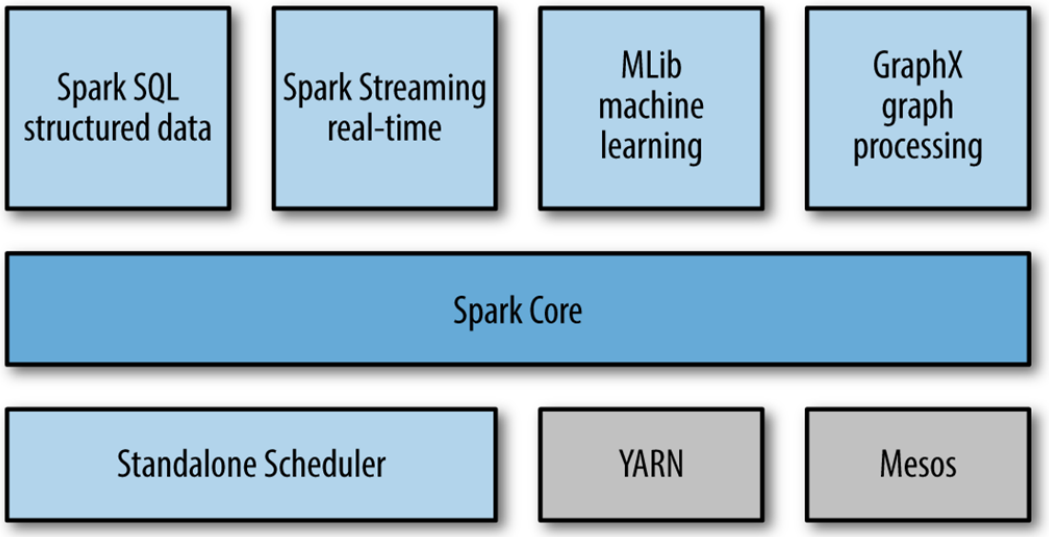


Figura 5 Componenti principali di Hadoop Spark

Spark richiede un gestore cluster e un sistema di storage distribuito. Per la gestione cluster spark supporta standalone (nativo spark), hadoop YARN o Apache Mesos. Per il sistema di storage distribuito può interfacciarsi con Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift, Amazon S3 o anche con una soluzione custom che può essere implementata. Spark supporta anche una modalità locale pseudo-distribuita, di solito utilizzata per scopi di sviluppo o di testing, dove lo storage distribuito non è richiesto e può essere utilizzato il file system locale.

Il progetto Spark contiene diversi componenti strettamente integrati (vedi Figura 5). Il “core” di spark è un motore computazionale che è responsabile della schedula, della distribuzione e del monitoraggio di applicazioni che consistono in più task computazionali tra diverse macchine.

Dato che il motore principale di Spark è sia veloce sia polivalente, alimenta più componenti di livello superiore specializzati per diversi carichi di lavoro, come ad esempio SQL o machine learning. Questi componenti sono stati progettati per interagire strettamente, si possono quindi combinare insieme come librerie di un progetto software.

Una filosofia di integrazione ha diversi vantaggi. In primo luogo, tutte le librerie e i componenti di livello superiore beneficiano dei miglioramenti dei componenti di livello più basso. Ad esempio, quando il motore di

base di spark aggiunge librerie di ottimizzazione anche SQL e machine learning ne beneficiano automaticamente. In secondo luogo, i costi associati alla gestione dello stack sono ridotti al minimo, perché invece di eseguire più sistemi di software indipendenti, se ne deve eseguire solo uno.

Questo significa anche che ogni volta che un nuovo componente viene aggiunto allo stack di Spark, ogni organizzazione che utilizza Spark sarà immediatamente in grado di provarlo. Questo cambia il costo di provare nuovi tipi di analisi.

Infine, uno dei più grandi vantaggi di una stretta integrazione è la capacità di creare applicazioni che combinano perfettamente diversi modelli di elaborazione. In Spark è possibile, ad esempio, scrivere un programma che utilizza machine-learning per classificare i dati in tempo reale alimentato da fonti di streaming. Allo stesso tempo, gli analisti possono interrogare i dati risultanti, anche in tempo reale, tramite SQL. Allo stesso modo si può accedere agli stessi dati attraverso una shell python e per tutto il tempo, il team IT deve mantenere un solo sistema.

1.2.1 Componenti

Spark Core

Il nucleo di Spark contiene le funzionalità di base, inclusi i componenti per la pianificazione delle attività, la gestione della memoria, il recupero dei guasti, l'interazione con i sistemi di storage e altro ancora. Il core è anche la sede di API che definiscono dataset distribuiti e resilienti (RDDs), che sono la principale astrazione di programmazione in Spark. Gli RDD rappresentano un'insieme di oggetti distribuiti attraverso molti nodi di calcolo che possono essere manipolati in parallelo. Il core fornisce molte API per la creazione e la manipolazione di queste collection.

Spark Sql

Spark SQL è un pacchetto per lavorare con dati strutturati. Esso permette l'interrogazione dei dati attraverso SQL così come la variante *Apache Hive* di SQL chiamata Hive Query Language (HQL) e supporta molte

fonti di dati, comprese le tabelle Hive, Parquet e JSON. Oltre a fornire un'interfaccia SQL per Spark, Spark SQL consente agli sviluppatori di mescolare query SQL con le manipolazioni di dati programmatici sostenuti da RDD in Python, Java e Scala, il tutto in una singola applicazione, combinando così SQL con analisi complesse.

Spark Streaming

Spark Streaming è un componente Spark che consente l'elaborazione di flussi di dati live. I flussi possono essere file di log generati da web server di produzione, le code di messaggi contenenti gli aggiornamenti di stato inseriti dagli utenti di un servizio web ecc ecc.

Spark Streaming fornisce API per la manipolazione di stream di dati che sono strettamente correlate alle API RDD di Spark Core, rendendo quindi facile per il programmatore imparare il progetto e passare tra le applicazioni che manipolano i dati memorizzati nella memoria, sul disco, o in arrivo in tempo reale.

MLib

Spark viene fornito con una libreria contenente le principali funzionalità di Machine-Learning (ML) chiamata Mllib. Essa offre diversi tipi di algoritmi di apprendimento automatico, compresa la classificazione, la regressione, il clustering e il collaborative filtering. Supporta anche funzionalità quali la valutazione del modello e l'importazione dei dati. Tutti questi metodi sono stati progettati per essere eseguiti attraverso un cluster.

GraphX

GraphX è una libreria per la manipolazione dei grafi (es: grafo delle amicizie in un social network) e esegue computazioni parallele sui grafi. Anche GraphX estende le Api RDD di Spark, permettendo la creazione di grafi orientati con proprietà arbitrarie che fanno riferimento a ogni vertice e bordo. Questo pacchetto fornisce anche diverse operazioni per la manipolazione del grafo e una libreria con i principali algoritmi.

Cluster Managers

Al suo interno, Spark è stato progettato per scalare in modo efficiente da uno a diverse migliaia di nodi di calcolo. Per raggiungere questo obiettivo, massimizzando la flessibilità, Spark può essere eseguito su una

varietà di *cluster manager*, tra cui Hadoop YARN, Apache Mesos, e un semplice gestore cluster incluso in Spark chiamato Standalone Scheduler [9].

1.2.2 Architettura

Le applicazioni spark possono essere eseguite come un set indipendente di processi su un cluster, coordinate dall'oggetto *SparkContext* nel programma principale (chiamato *driver program*).

Nello specifico, per essere eseguito su un cluster, lo *SparkContext* può connettersi con diversi tipi di *cluster manager* (lo stesso cluster manager di spark standalone, Mesos o YARN) che allocano risorse alle applicazioni. Una volta connesso, Spark acquisisce degli *executors* sui nodi del cluster, che sono processi che eseguono computazioni e immagazzinano dati per le nostre applicazioni. Successivamente, viene inviato il codice della nostra applicazione (definito come file JAR o python passati allo SparkContext) agli *executor*. Infine lo SparkContext invia i *task* agli executor per essere eseguiti. In Figura 6 si può vedere graficamente tutto il processo.

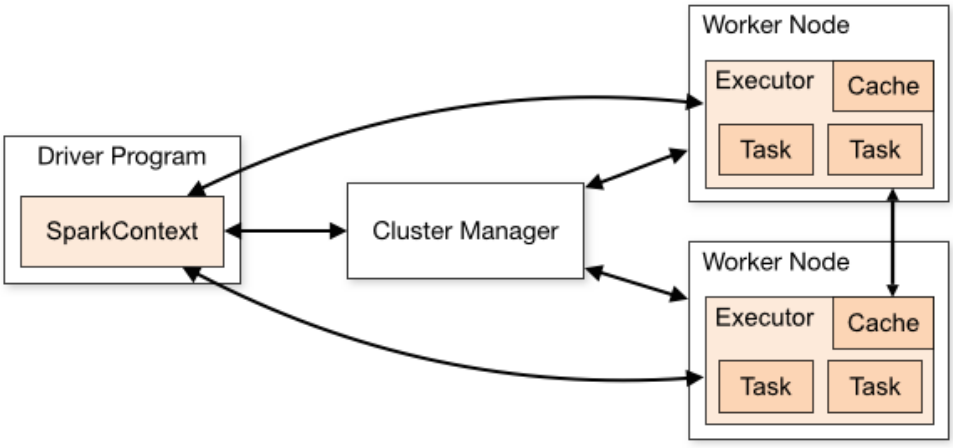


Figura 6 Architettura di Hadoop Spark

Questo tipo di architettura ci permette di fare alcune considerazioni riguardo l'esecuzione di uno o più programmi in spark:

- Ogni applicazione ottiene i suoi executor, che rimangono per tutta la durata dell'applicazione ed eseguono i task su più thread.

Questo ha il vantaggio di isolare le applicazioni le une dalle altre, sia dal lato programmazione che dal lato esecuzione (attività diverse eseguite da JVM diverse). Tuttavia, ciò significa anche che i dati non possono essere condivisi tra applicazioni Spark diverse (istanze di SparkContext) senza scriverli su di un sistema di memorizzazione esterno.

- Spark è indifferente al cluster manager sottostante. Finché può acquisire processi executor e farli comunicare è relativamente facile da eseguire anche su un cluster manager che supporta altre applicazioni (ad esempio Mesos o YARN).
- Il programma driver deve rimanere in ascolto per accettare connessioni in ingresso dai suoi executor per tutto il suo ciclo di vita. Come tale il programma driver deve essere indirizzabile e raggiungibile su rete dai worker nodes.
- Dato che il driver schedula i task sul cluster, deve essere eseguito vicino ai worker node, preferibilmente sulla stessa rete locale. Se si vogliono inviare le richieste al cluster in modalità remota, è meglio aprire una RPC al driver e inviare le operazioni da vicine che eseguire un driver lontano dai worker node.

[10]

1.2.3 Scheduling

Spark dispone di numerosi servizi per la schedulazioni di risorse tra computazioni. In primo luogo ogni applicazione Spark (quindi istanza di SparkContext) gestice un insieme indipendente di processi executor. I cluster Manager su cui spark viene eseguito forniscono servizi per la schedula tra le applicazioni. In secondo luogo, al'interno di ogni applicazione Spark, più job (azioni Spark) possono essere in esecuzione contemporaneamente se vengono eseguiti da diversi thread. Questo è comune se l'applicazione serve richieste dalla rete. Spark include un fair scheduler e un schedulatore di risorse all'interno di ogni SparkContext.

Schedulazione Tra Applicazioni

Quando viene eseguita su un cluster, ogni applicazione Spark ha un set indipendente di executor JVM che eseguono task e salvano dati per

l'applicazione. Se più utenti hanno bisogno di condividere un cluster, ci sono diverse opzioni per l'allocazione, dipende dal cluster manager su cui si sta lavorando.

L'opzione più semplice, disponibile su tutti i cluster manager, è il *partizionamento statico* delle risorse. Con questo approccio, ad ogni applicazione è dato un massimo numero di risorse di cui può disporre e di cui ha il controllo per tutta la sua durata. Questo è l'approccio utilizzato in standalone e in YARN, così come nel *coarse-grained Mesos mode*. L'allocazione delle risorse può essere configurato come segue, in base al tipo di cluster:

- **Standalone mode**: di default, le applicazioni in esecuzione su standalone mode cluster vengono eseguite in ordine FIFO (first-in-first-out) e ogni applicazione cercherà di utilizzare tutti i nodi disponibili. È possibile limitare il numero di nodi che utilizza un'applicazione impostando la proprietà di configurazione *spark.core.max*, o modificare il valore predefinito *spark.deploy.defaultCores.Finally*. in aggiunta al controllo dei core, il settaggio *spark.executor.memory* controlla il suo utilizzo della memoria.
- **Mesos**: per usare il partizionamento statico su Mesos, bisogna settare la proprietà *spark.mesos.coarse* a true e opzionalmente *spark.cores.max* per limitare le risorse per ogni applicazione come nello standalone mode. Si può settare anche *spark.executor.memory* per controllare la memoria degli executor.
- **YARN**: l'opzione *--num-executor* nel client YARN controlla quanti executor allocerà al cluster, mentre *--executor-memory* e *--executor-cores* controlla le risorse per executor

Una seconda opzione disponibile su mesos è il *dynamic sharing* dei CPU cores. In questo modo, ogni applicazione spark ha ancora un numero fisso e indipendente di allocazione di memoria (settando *spark.executor.memory*), ma quando un applicazione non sta eseguendo un task su una macchina, altre applicazioni possono eseguire task su quei core. Questa modalità è utile quando ci si aspetta un grande numero di applicazioni non particolarmente attive, come le sessioni di shell di utenti separati. Tuttavia, si può avere un rischio di latenza non

prevedibile, in quanto può passare del tempo per riottenere i core di un nodo quando si vogliono riutilizzare.

Si noti che nessuna delle modalità presentate attualmente fornisce al condivisione della memoria tra le applicazioni. Se si desidera condividere i dati in questo modo, si consiglia di eseguire una singola applicazione server che può servire più richieste interrogando le stesse RDDs.

Allocazione dinamica delle risorse

Spark fornisce un meccanismo per equilibrare dinamicamente le risorse delle applicazioni basato sul carico di lavoro. Questo significa che un'applicazione può restituire risorse al cluster se non sono usate da tempo e richiederle ancora quando ci sarà bisogno. Questa feature è particolarmente utile se diverse applicazioni condividono risorse nel cluster spark [11].

1.2.4 RDD Abstraction

L'astrazione che utilizza il nucleo di spark per lavorare e interagire con i dati sono i **Resilient Distributed datase (RDD)**. Un RDD è semplicemente una collezione di elementi. Un programma in spark non è altro che la creazione di nuovi RDD, la trasformazione di RDD esistenti o l'invocazione di operazioni su RDD per calcolare un risultato. Al suo interno, Spark distribuisce automaticamente i dati contenuti nell'RDD attraverso il cluster e parallelizza le operazioni da eseguire su di essi.

Ogni RDD è divisa in più partizioni, che possono essere computate su nodi differenti del cluster. RDD può contenere ogni tipo di oggetto Python, Java o Scala incluse classi definite dagli utenti.

Un utente può creare RDD in due modi: caricando dataset esterni oppure distribuendo una collezione di oggetti. Una volta creato, l'RDD offre due tipi di operazioni: *trasformation* e *action*.

Le ***trasformation*** costruiscono nuove RDD da RDD precedenti. Una trasformazione comune è il filtraggio dei dati che corrispondono ad un determinato pattern, per esempio nel caso di RDD di tipo testo si

vogliono prendere in esame tutte le RDD che contengono una determinata parola.

Le ***action***, dall'altro lato, computano un risultato basato sugli RDD che può essere poi consegnato al programma principale oppure salvato su un sistema di memorizzazione (HDFS). Un esempio di azione è la funzione *first()* che restituisce il primo elemento di un RDD.

Trasformation e *action* si differenziano per il modo in cui Spark calcola gli RDD. Anche se è possibile definire nuovi RDD in qualsiasi momento, Spark li calcola in *lazy-mode* cioè quando per la prima volta sono utilizzati in una action. Questo approccio potrebbe sembrare inusuale all'inizio, ma ha senso quando si lavora con i Big Data. Se ogni volta che si esegue una trasformation Spark dovesse caricare e memorizzare ogni volta tutti gli elementi dell'RDD sprecherebbe molto spazio di archiviazione. Invece, una volta che Spark vede tutta la catena delle trasformation, si può calcolare solo i dati necessari per il suo risultato. Ad esempio se la prima action che si esegue sul file è *first()* Spark lo analizza fino alla prima linea corrispondente, e non legge l'intero file calcolando tutte le trasformation.

In definitiva, le RDD di Spark sono di default ricalcolate ogni volta che si esegue un action su di loro (lazy-evaluation). Se si desidera riutilizzare un RDD in più azioni, è possibile chiedere a Spark di mantenerla usando *RDD.persist()*. Possiamo chiedere a Spark di mantenere i nostri dati in un numero di posti diversi. Una volta computati la prima volta, Spark memorizzerà il contenuto delle RDD in memoria (partizionato su più macchine del cluster) e lo riuserà in azioni future.

Infine, se viene derivata una nuova RDD da una esistente attraverso una trasformation, Spark tiene traccia del set delle dipendenze tra le differenti RDD attraverso un grafo chiamato *lineage graph*. Spark utilizza queste informazioni per computare ogni RDD a richiesta e per recuperare eventuali dati persi se parte delle RDD sulle quali è stato chiamato il *persist* vengono perse. In Figura 7 viene mostrato un esempio di *lineage graph* [9].

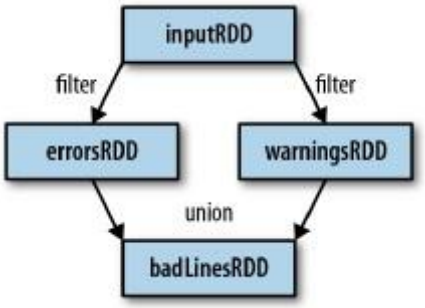


Figura 7 Esempio di lineage graph

1.2.5 Componenti di esecuzione: Job, Task e Stage

Spark trasforma la sua rappresentazione logica delle RDD e delle loro partizioni in una pianificazione fisica unendo più operazioni in task.

Quando viene eseguita una operazione di tipo *action* su un RDD spark crea un piano di esecuzione fisico per computare le RDD necessarie per eseguire la action. Tutte le partizioni dell'RDD devono essere materializzate e poi trasferite al programma driver. Lo scheduler di Spark inizia all'ultima RDD che deve essere calcolata e lavora a ritroso per trovare ciò che si deve calcolare. Sviluppa un piano fisico attraverso la ricorsione a tutti i *parent* della RDD di partenza. Nel caso più semplice lo scheduler realizza un ***stage*** computazionale per ogni RDD del grafo dove lo stage ha dei ***task*** per ogni partizione in quella RDD.

In casi più complessi, il set fisico di stage non sempre corrisponderà in modo esatto (1:1) con il grafo delle RDD. Questo può succedere quando lo scheduler esegue pipelining o collassa più RDD in un singolo stage.

Lo scheduler interno di Spak può anche decidere di troncare il lineage del grafo RDD se su una RDD esistente è stato eseguito un *persist*. Spark in questo caso può iniziare la computazione in base alla RDD persistente. Un secondo caso in cui può succedere che venga troncato il lineage si ha quando una RDD è già stata materializzata come side effect anche se non è stato fatto un persist esplicito.

La Figura 8 mostra come viene trasformato un grafo RDD in stage fisici.

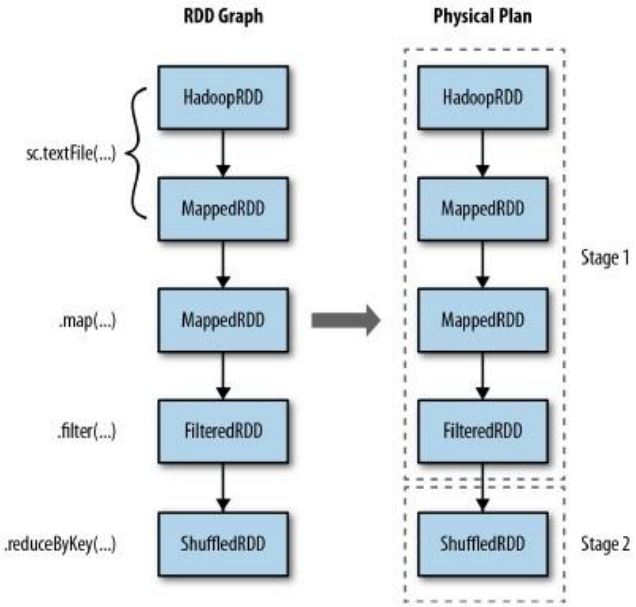


Figura 8 Trasformazione del grafo RDD in stage fisici

L'insieme degli stage prodotti per una particolare action viene chiamato *job*. Ogni volta che viene chiamata una qualsiasi action viene creato un job composto da uno o più stage.

Una volta che è stato definito il grafo degli stage, vengono creati e inviati i task ad uno scheduler interno, che varia a seconda del deployment mode che si sta utilizzando. Gli stage nel piano fisico possono dipendere gli uni dagli altri, basati appunto sul lineage dell'RDD, quindi dovranno essere eseguiti in un ordine specifico.

Uno stage fisico manda in esecuzione task che eseguono la stessa cosa ma su una specifica partizione dei dati. Ogni task internamente esegue gli stessi step:

1. Recupera l'input, sia da archiviazione dei dati (se la RDD è in input) o da un RDD esistente (se è stata memorizzata nella cache).
2. Esegue le operazioni necessarie per computare la/le RDD che rappresenta.
3. Scrive in output su archiviazione esterna o restituisce di nuovo al driver (nel caso l'RDD calcolata sia l'RDD finale)

Riassumendo, le seguenti fasi si verificano durante l'esecuzione di Spark:

Il codice utente definisce un DAG (directed acyclic graph) di RDD: le operazioni sulle RDD definiscono nuove RDD che si riferiscono alle RDD padri, in tal modo viene definito un grafo.

Le azioni forzano la traduzione del DAG in piani di esecuzione fisici: quando viene chiamata una action l'RDD dovrà essere computata. Questo richiede la computazione anche delle RDD padri. Lo scheduler di spark esegue un *job* per computare tutte le RDD richieste. Questo job può avere uno o più *stage* che sono composti da *task*. Ogni stage corrisponderà a una o più RDD del DAG. Un singolo stage può corrispondere a più RDD in base al pipeling.

I task sono schedulati e eseguiti su un cluster: gli stage sono processati in ordine, con task individuali lanciati su segmenti dell'RDD. Una volta che lo stage finale termina con il job, la action è completa [9].

1.2.6 DataFrame

Un DataFrame è una collezione distribuita di dati organizzata in colonne nominali. È concettualmente equivalente ad una tabella in un database relazione o ad un data frame in R/python, ma con una maggiore ottimizzazione al suo interno.

Un DataFrame può essere costruito da una vasta gamma di fonti come: i file di dati strutturati, tabelle in Hive, database esterni o RDD esistenti.

Una volta costruito, il DataFrame fornisce un domain-specific language per la manipolazione dei dati distribuiti. Si può anche incorporare SQL mentre si sta lavorando con i DataFrame usando Spark SQL.

Così come gli RDD, anche i DataFrame vengono valutati con una lazy evaluation, vale a dire che il calcolo avviene solo quando è richiesta un'azione (ad esempio la visualizzazione del risultato, salvare l'output ecc). Questo permette alla loro esecuzione di essere ottimizzata. Inoltre tutte le operazioni sui DataFrame vengono anche automaticamente parallelizzate e distribuite sul cluster.

Prima di ogni computazione su un DataFrame, il *Catalyst optimizer* inserisce le operazioni che dovranno essere eseguite per realizzare il

DataFrame in un piano fisico di esecuzione. Dato che l'optimizer riesce a comprendere la semantica delle operazione e la struttura dei dati, può prendere decisioni "intelligenti" per velocizzare la computazione [12].

2 Analisi di dati genomici

All’interno di questo capitolo verrà fatta una breve introduzione alla genomica, al ruolo che svolge all’interno della bioinformatica e alle potenzialità che potrà avere in futuro.

In particolare verrà fatta una panoramica sulle pubblicazioni e sui progetti che riguardano le tecnologie OLAP e le tecnologie di data warehouse in campo genomico.

Infine verrà affrontato il database preso in esame per questo lavoro di tesi, il database TCGA. In particolare verranno analizzati tutti i data type presenti al suo interno e i vari data level in cui sono organizzati. Inoltre verranno valutati i differenti metodi in cui si possono reperire i dati.

2.1 Obiettivi e problematiche

La **genomica** è una branca della *biologia molecolare* che si occupa dello studio del genoma degli organismi viventi. In particolare si occupa della struttura, contenuto, funzione ed evoluzione del genoma. È una scienza che si basa sulla bioinformatica per l'elaborazione e la visualizzazione dell'enorme quantità di dati che produce.

Alla base della genomica sono i metodi della biologia molecolare, quali i metodi di clonaggio dei geni e di sequenziamento del DNA. Conoscere l'intero genoma degli organismi permette di assumere un approccio nuovo in silico alla ricerca biologica. Questo presenta molti vantaggi: un esempio è la maggior facilità del trasferimento delle conoscenze su un organismo ad un altro, tramite la ricerca di geni omologhi. Un altro vantaggio è chiaramente osservabile per esempio in campo biomedico: molte malattie sono complesse, determinate da molti geni, come i tumori, e la conoscenza dell'intero genoma permette di identificare con più facilità i geni coinvolti e osservare come questi interagiscono nel loro background genetico.

Grazie alle scoperte derivanti dal sequenziamento del genoma umano è nata una nuova branca definita genetica personalizzata e derivante dalle applicazioni delle conoscenze genetiche in medicina e nella pratica clinica. è ora infatti possibile eseguire degli studi predittivi sull'incidenza

di una data patologia su un campione o su un individuo rispetto alla popolazione generale per definire il rischio di sviluppare quella patologia.

La genomica nacque negli anni 80, quando furono prese le prime iniziative per il sequenziamento di interi genomi. Una data di nascita si può probabilmente far coincidere con il sequenziamento completo del primo genoma, nel 1980: si trattava del genoma di un virus, il fago Φ-X174. Il primo sequenziamento del genoma di un organismo vero e proprio fu completato nel 1995 e si trattava di un batterio, Haemophilus influenzae, con un genoma di notevoli dimensioni (1,8 milioni di paia di basi). Da allora i genomi "completati" aumentano esponenzialmente. La prima pianta il cui genoma è completamente noto nella sua sequenza è stata Arabidopsis thaliana.

Con grande rilievo fu data dalla stampa non specialistica la notizia del sequenziamento del genoma umano, nel 2001. Un progetto pubblico di sequenziamento nacque nel 1986 sotto il nome di progetto genoma umano. Nonostante l'enorme quantità di soldi spesi (il finanziamento maggiore era statunitense, pari a tre miliardi di dollari) da vari stati per i finanziamenti pubblici, questo fu pesantemente sconfitto dalla strategia innovativa di sequenziamento di un'azienda privata, la Celera Genomics, che riuscì a completare (nel 2000) il sequenziamento del genoma umano in una frazione del tempo e dei costi rispetto al progetto pubblico (300 milioni di dollari), facendo scoppiare grandi polemiche e cambiando radicalmente l'approccio seguito dai progetti pubblici. La sequenza fu pubblicata nel 2001 in un articolo su Nature, che combinava i risultati di entrambi i progetti; si trattava di una bozza pari al 90% della sequenza e ancora con notevoli probabilità di errori. Una sequenza accurata al 99,99% e pari al 99% del genoma umano è stata pubblicata nel 2003.

Tra gli obiettivi che si pone la genomica vi è dunque l'allestimento di complete mappe genetiche e fisiche del DNA degli organismi viventi, proseguendo con il suo completo sequenziamento. La sequenza del DNA viene poi annotata, ovvero vengono identificati e segnalati tutti i geni e le altre porzioni di sequenza significative, insieme a tutte le informazioni conosciute su tali geni. In questo modo è possibile ritrovare in maniera organizzata ed efficace le informazioni in appositi database,

normalmente accessibili via Internet gratuitamente. Grazie al sequenziamento di diversi genomi è nata la genomica comparativa, che si occupa del confronto tra i genomi di diversi organismi, nella loro organizzazione e sequenza [13].

Il problema fondamentale che si trova ad affrontare oggi la ricerca biologica è quello di disporre di una quantità immensa e crescente di dati, che viene prodotta empiricamente dagli studi chimici, biochimici e genetico-molecolari, e che però non rappresenta una effettiva conoscenza sul modo di funzionare dei sistemi biologici.

Per gestire questa massa di informazioni biologiche in continuo aumento e per analizzarle in modo da trovare delle relazioni che facciano emergere qualche nuovo principio organizzativo o funzionale della vita è nata la bioinformatica. La bioinformatica include quindi l'integrazione e l'esplorazione delle sempre più estese banche dati di interesse biologico [14].

2.2 Stato dell'arte

Le tecnologie OLAP e di Data Warehousing sono abbastanza mature e largamente utilizzate in contesti di business, tuttavia non esistono molti progetti sulle loro applicazioni in ambito genomico. Nella letteratura, è stata proposta solo qualche applicazione di datawarehouse per dati biologici. Qualcuno di questi affronta la sfida di integrare sorgenti di dati eterogenee al fine di produrre un database riconciliato ma non fornisce uno schema multidimensionale subject-oriented. In questa sessione ci focalizzeremo su quelle applicazioni che trattano di analisi OLAP, modellazione data warehouse, e tecniche di data mining in campo genomico. Sebbene alcune pubblicazioni consiglino l'uso di OLAP come strumento per scoprire le informazioni, solo in pochi casi è stato effettivamente impiegato.

Nel paper [15] gli autori usano le tecnologie OLAP per analizzare i dati di espressione genomica, raccolti da test effettuati su radici di soia, al fine di scoprire informazioni per sviluppare coltivazioni di soia resistenti al parassita SNC. Nel paper [16] invece sono state trovate importanti

correlazioni tra modelli di cancellazione nel cromosoma Y e popolazioni di pazienti.

I modelli di data warehouse, concettuali e logici, rispetto a dati di tipo genomico, o più in generale rispetto a dati di tipo biomedico è un argomento non ben studiato e con solo qualche pubblicazione esistente. Nello studio [17] gli autori propongono modelli per 3 spazi di dati di espressioni genomiche (Sample, annotation e gene expression) basati su schemi a stella o a fiocco di neve. Uno studio ancora più esaustivo sul modello multidimensionale biomedico è presentato nel paper [18] insieme a un nuovo schema chiamato BioStar. Gli autori sostengono che attraverso questo schema si possono affrontare le sfide del dominio genomico, come le strutture a rapida evoluzione, i dati imprecisi e incompleti etc.

Negli ultimi anni, diverse applicazioni che estraggono informazioni interessanti su banche dati genomiche sono state ideate, tuttavia, l'importanza di approcci di data mining nel contesto genomico si incrementerà in futuro.

Il data mining genomico è stato applicato per affrontare diversi problemi derivanti dalla complessità dei volumi dei dati genomici, come le grandi dimensioni dei dadi biologici, il rumore dei dati, il volume dei dati da analizzare, domini parzialmente sconosciuti e complessi. Nello studio [19] sono state utilizzate tecniche di data mining con successo per inferire la prognosi e le informazioni dei pazienti chemioterapici dalla loro espressione genomica univoca. Sono state poi utilizzate tecniche di profilazione per l'espressione di RNA nello studio [20] in modo che i modelli di espressione possano essere misurati su campioni di pazienti.

Infine le tecniche di estrazione sono state applicate per superare l'eterogeneità di insiemi di dati provenienti da diversi laboratori, in modo da consentire indagini difficilmente ottenibili con metodi tradizionali nello studio [21].

Anche se le tecniche di data mining sono spesso considerate più potenti di quelle tradizionali OLAP, i due approcci dovrebbero essere considerati complementari, piuttosto che in competizione. Infatti, mentre l'analisi OLAP permette una navigazione tra i dati user driven e

semplice da capire il data mining richiede utenti esperti e i suoi risultati sono spesso difficili da interpretare.

2.3 Il database TCGA

La sorgente dati proviene dal **National Cancer Institute** (NCI) che fa parte del **National Institute of Health** (NIC) all'interno del Dipartimento della Salute e dei Servizi Umani degli Stati Uniti. L'NCI coordina l'U.S. National Cancer Program e sostiene la ricerca, la formazione, la diffusione dell'informazione sanitaria e altre attività collegate alle cause, alla prevenzione, alla diagnosi e al trattamento del cancro.

Il Cancer Genome Atlas (TCGA) è stato avviato come progetto pilota nel 2006 attraverso investimenti sostenuti dal National Cancer Institute e dall'National Human Genome Research Institute (NHGRI). Il TCGA ha lo scopo di creare un catalogo delle mutazioni genetiche responsabili del cancro, usando sequenze genomiche e bioinformatica. Rappresenta uno sforzo nella guerra al cancro applicando tecniche high-troughput di analisi del genoma per migliorare la capacità di diagnosticare, trattare e prevenire il cancro attraverso una migliore comprensione delle basi genetiche di questa malattia.

Il progetto utilizza 500 campioni di pazienti, molto più della maggior parte degli altri studi genomici, e usa differenti tecniche per analizzare i campioni dei pazienti.

Il Cancer Genome Atlas (TCGA) migliora in maniera globale e coordinata la nostra comprensione delle basi molecolari del cancro attraverso l'applicazione di tecnologie di analisi del genoma.

2.3.1 Portale Dati

Il **TCGA Data Portal** fornisce una piattaforma per i ricercatori per cercare, scaricare e analizzare insiemi di dati generati da TCGA. Esso contiene informazioni cliniche, dati di caratterizzazione genomica e analisi di alto livello sulle sequenze dei genomi tumorali.

2.3.2 Data Level

Data Level è un metodo di categorizzazione usato all'interno della rete TCGA per facilitare i ricercatori nella comunicazione e nella localizzazione dei dati di interesse. I Data Level vengono assegnati per ogni data type, piattaforma o centro. Ci sono 4 livelli di dati:

Data Level	Level Type	Description
1	Grezzi	Dati di basso livello per singolo campione Non normalizzati
2	Processati	Dati normalizzati per singolo campione Interpretati per presenza o assenza di anomalie molecolari specifiche
3	Segmentati/Interpretati	Aggregazione di dati processati da singolo campione Raggruppati per formare regioni contigue più grandi
4	Regione di interesse	Associazione quantitativa lungo classi di campioni Associazioni basate su due o più Anomalie molecolari Caratteristiche del campione Variabili cliniche

Ogni piattaforma è in grado di produrre più Data Type. Per capire la categorizzazione è importante chiarire la relazione tra data type e data level.

Ogni data type è associato ad insiemi di dati che si estendono a uno o più data level. Ogni centro o piattaforma può avere un concetto leggermente diverso di data level dipendente dai data type e dagli algoritmi utilizzati per l'analisi all'interno della specifica piattaforma.

2.3.3 Data Type

Ecco un elenco di tutti i tipi di dati che si possono trovare all'interno del portale

Clinical Data

Sottotipi: Clinical Data (.xml), Biospecimen Data (.xml), Pathology Reports (.pdf).

Descrizione: Sono tipi di dato che si possono trovare nei data level 1 e nei data level 2. Il dizionario dei dati BCR descrive gli elementi dei dati clinici e dei campioni biologici in TCGA.

Images

Sottotipi:Diagnostic image (.svs), Tissue image (.svs), Radiological image (.dcm).

Descrizione: Sono tipi di dato presenti solo nei data level 1.

Microsatellite Instability (MSI)

Drescrizione: Tipi di dato presenti ai livelli 1 e 3 in formato .fsa e .txt. Dati ad accesso controllato.

DNA Sequencing

Sottotipi: sequenza esomica(.bam ad accesso controllato), sequenza genomica(.bam ad accesso controllato), sequenza tracce(.sfc), mutazioni(.maf)

Descrizione: ogni file bam ha associato un file xml con i relativi metadati. I file sulle mutazioni non hanno invece uno standard.

miRNA Sequencing

Sottotipi:sequenze di miRNA(.bam ad accesso controllato), miRNA, Isoformi

Descrizione: Utilizzati nei livelli 1 e 3.

Protein Expression

Descrizione: Tipi di dato presenti ai livelli 1,2,3 in formati .tiff o .txt.

mRNA Sequencing

Sottotipi:mRNA sequence (.BAM ad accesso controllato), Esomi (.txt), Geni (.txt), Splice Junction(.txt), Isoform (.txt)

Descrizione: tipi di dato presenti ai livelli 1 e 3, la descrizione di questi file è contenuta nel DESCRIPTION file dell'archivio.

Total RNA Sequencing

Sottotipi:mRNA sequence (.BAM ad accesso controllato), Esomi (.txt), Geni (.txt), Splice Junction(.txt), Isoform (.txt)

Descrizione: tipi di dato presenti ai livelli 1 e 3, la descrizione di questi file è contenuta nel DESCRIPTION file dell'archivio.

Array-based Expression

Sottotipi:Gene (.txt), Esomi (.cel file binario e .txt), miRNA(.txt)

Descrizione: tipi di dato presenti ai livelli 1, 2 e 3. La descrizione di questi file è contenuta nel DESCRIPTION file dell'archivio. Altre informazioni possono essere contenute nell'Array design file di ogni piattaforma

DNA Methylation

Sottotipi: Bisulfite sequencig (.bam, .vcf, .bed accesso controllato), Array based (.idat, .txt)

Descrizione: tipi di dato presenti ai livelli 1, 2 e 3. La descrizione di questi file è contenuta nel DESCRIPTION file dell'archivio.

Copy Number

Sottotipi: SNP(.cel, .txt accesso controllato), Array (.txt, .tsv, .mat), Low-Pass DNA Sequencing (.bam, .vcf accesso controllato)

Descrizione: tipi di dato presenti ai livelli 1, 2 e 3. La descrizione di questi file è contenuta nel DESCRIPTION file dell'archivio. Altre informazioni possono essere contenute nell'Array design file di ogni piattaforma.

2.3.4 Come ottenere i Dati

Il portale dati TCGA fornisce 4 possibilità di accesso ai dati:

Metodo	Descrizione	Limitazioni
Data Matrix	Permette agli utenti di selezionare e scaricare sottoinsiemi di dati basati su specifici criteri inclusi il centro la piattaforma e il data type	Il Data Matrix fornisce soltanto l'ultima revisione per ogni archivio, le altre revisioni sono presenti in bulk download o attraverso l'accesso http. Inoltre non permette la ricerca attraverso più malattie.
Bulk Download	Consente agli utenti di cercare e scaricare archivi di dati caricati dai centri TCGA	
Access http Directories	Consente agli utenti di accedere alle directory http in cui sono memorizzati gli archivi di dati	
File Search	Consente agli utenti di filtrare e scaricare i file utilizzando criteri come la malattia, categoria, data level. A differenza di Data Matrix, la ricerca file TCGA permette la ricerca su più malattie	La ricerca file fornisce soltanto l'ultima revisione per ogni archivio, le altre revisioni sono presenti in bulk download o attraverso l'accesso http

2.3.5 Mole di dati

Attraverso l'accesso ai file tramite l'applicativo File Search è possibile fare una prima stima della mole di dati presente nel database TCGA:

Data Type	Size (GB)
Clinical Data	3795,499
DNA Mutation	5068
mRNA Expression	288,472

miRNA Expression	4,182	
DNA Copy Number	2,506	
DNA Methylation	581,266	
Protein Expression	82,43	
Others	2251,616	
Somma	12073,971	GB
	11,7909873	TB

2.4 Il linguaggio GMQL e il Genome Space

Il sequenziamento di nuova generazione (Next Generation Sequencing - NGS) permette analisi high-troughput del sequenziamento del genoma (DNA-ss), di transcriptome profiling (RNA-seq), della valutazione dell'interazione DNA-Proteina (ChIP-Seq) e della caratterizzazione epigenome (ChIP-Seq,BS-Seq,DNase-ss).

Miglioramenti continui delle tecnologie NGS in qualità, costi dei risultati e tempo di sequenziamento stanno portando in breve alla possibilità di sequenziare un intero genoma umano in pochi minuti per un costo di meno di 1000$ [22] [23].

Per questo motivo stanno emergendo diversi progetti sul sequenziamento su larga scala. Grazie a una così alta disponibilità di differenti tipi di dati NGS provenienti da numerosi genomi individuali, è ora possibile analizzare più istanze di differenti caratteristiche genomiche simultaneamente, così da caratterizzare il loro ruolo funzionale e chiarire i fenomeni genetici e epigenetici. Questo richiede una nuova generazione di sistemi informatici e linguaggi per interrogare dataset eterogenei. Richiede anche la ricerca di strategie di calcolo parallelo per distribuire la computazione attraverso cloud di calcolatori, per ottenere scalabilità e migliori prestazioni.

Sebbene la maggior parte degli strumenti di analisi bioinformatica corrente non supporta la parallelizzazione, ultimamente sono stati adottati anche in bioinformatica sistemi di cloud computing paralleli ad alte prestazioni, basati principalmente sui framework Hadoop [24] e MapReduce [25] che sono tipicamente usati in altre aree.

Pochi tool specifici sono stati implementati [26], in particolare *BioPig* [27] è stato proposto per il trattamento efficace dei programmi di bioinformatica che utilizzano *Pig Latin* [28], un linguaggio ad alto livello per l'elaborazione dei dati batch. [29]

2.4.1 Genometric Query Language (GMQL)

GMQL è ispirato da *Pig Latin* [30] che combina lo stile dichiarativo ad alto livello di SQL con lo stile procedurale di più basso livello di map-reduce [31] [32]. I programmi Pig Latin sono compilati in piani fisici che poi vengono eseguiti su Hadoop [24] [33]. Gli utenti di Pig Latin specificano una sequenza di step dove in ogni step è specificata una sola transformazione dei dati ad alto livello.

Una query GMQL è espressa attraverso una sequenza di operazioni GMQL con la seguente struttura:

<variable> = operation(<parameters>) <variables>

Le operazioni vengono applicate a uno o più parametri e restituiscono una variabile risultato. I parametri sono specifici per ogni operazione ed includono i predicati, usati per fare select e join dei dataset o delle regioni.

Le operazioni sui metadati si applicano a un singolo set di dati ed hanno le seguenti caratteristiche generali:

- Filtro SELECT che lascia fuori alcuni campioni utilizzando un predicato sugli attributi dei metadati.
- AGGREGATE applica funzioni di aggregazione a valori delle regioni per ogni campione e aggiunge i risultati a nuovi attributi dei metadati.
- ORDER utilizza gli attributi dei metadati per ordinare i campioni e, eventualmente, per filtrare quelli in cima alla lista degli ordinati.

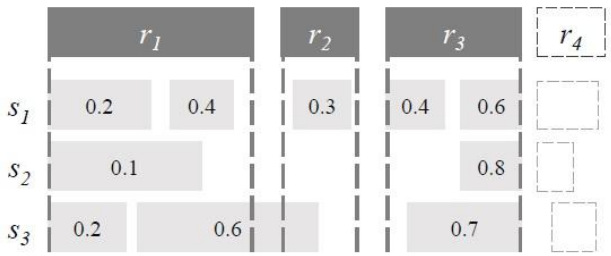
Tutte le operazioni appena descritte creano un nuovo dataset, i campioni che non vengono filtrati mediante operazioni sono compresi nell'insieme di dati risultato senza cambiare i loro identificatori e le loro regioni.

GMQL utilizza il Genomic Data Model (GDM) basato sulla nozione di regione genomica, che può estendersi su diverse basi, ogni regione può essere confrontata con milioni di altre regioni, in genere utilizzando proprietà metriche, inoltre GDM copre anche metadati della struttura arbitraria. GMQL fa parte di GenData 2020 [34], un sistema che supporta un layer di data warehouse per la memorizzazione di file di dati nel loro formato originale, fornendo adeguati livelli di privacy, e un layer di selezione dei dati, consentendo astrazioni sui formati di file e parallelizzazione [29].

2.4.2 Genome Space

Il genome space è un dataset in cui ad ogni coppia di regione di riferimento *r* e ad ogni campione di ingresso *s* è associato un insieme di valori numerici, ciascuno ottenuto applicando una funzione di aggregazione (ad esempio la media) per una caratteristica specifica *c* (ad esempio il p-value) delle regioni di *s* che si sovrappongono con *r*.

Un'operazione di Mapping è definita da una regione di riferimento, un campione di ingresso, la funzione di aggregazione selezionata e la caratteristica della regione usata per aggregare. Il genome space risultante da un'operazione di mapping viene materializzato come un insieme di file GTF e di relativi metadati, più precisamente, un file GTF e un file di metadati per ogni campione in ingresso. Nell'esempio raffigurato in Figura 9 si può vedere la rappresentazione di un mapping tra campioni e regioni e il relativo genome space risultante.



	r_1	r_2	r_3
s_1	0.3	0.3	0.5
s_2	0.1		0.8
s_3	0.4	0.6	0.7

(b)

Figura 9 Mapping tra le regioni di riferimento r1, r2, r3 e i campioni in imput s1, s2, s3 (a) e il genome space dove ogni cella è computata come la media dei valori nelle regioni corrispondenti (b)

3 Scoperta automatica di dipendenze funzionali approssimate

Nella gestione della qualità dei dati e nella rappresentazione della conoscenza le dipendenze funzionali giocano un ruolo davvero importante.

Il processo della ricerca di dipendenze funzionali all'interno di database si chiama dependency discovery e può essere molto utile per estrarre conoscenza in determinati contesti applicativi.

3.1 Definizione del problema

Nella teoria dei database relazionali, una dipendenza funzionale è un vincolo tra due set di attributi in una relazione nel database.

Data una relazione R, un set di attributi X in R si dice che determina funzionalmente un altro set di attributi Y, sempre in R, (scritto $X \rightarrow Y$) se, e solo se, ad ogni valore di X è associato precisamente un valore di Y; R allora soddisferà la dipendenza funzionale $X \rightarrow Y$.

In altre parole, un dipendenza FD: $X \rightarrow Y$ significa che i valori di Y sono determinati dai valori di X. Due tuple che condividono gli stessi valori di X dovranno necessariamente avere gli stessi valori di Y.

3.1.1 Dipendenze funzionali esatte

Sia $R = \{A_1, \dots, A_m\}$ uno schema di relazione di un database relazionale ed r una sua istanza, con $t[X]$ indichiamo la proiezione di una tupla t di r sul sottoinsieme di attributi $X \subseteq R$.

Definizione: Una **dipendenza funzionale** (**FD**) $X \rightarrow Y$ dove $X, Y \subseteq R$, è soddisfatta da un'istanza r se per qualsiasi coppia di tuple $t_1, t_2 \in r$, se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$.
Il sottoinsieme X è chiamato *parte sinistra* (lhs) o determinante, mentre Y è chiamato *parte destra* (rhs) o dipendente.

Le dipendenze così definite si dicono esatte poiché il confronto tra le proiezioni è di tipo esatto e la condizione deve essere valida per tutte le

tuple del database. Una FD viene detta **minimale** se togliendo un attributo lhs, questa non è più tale.

3.1.2 Dipendenze funzionali approssimate

Esistono vari tipi di dipendenze funzionali approssimate che derogano su qualcuno degli aspetti che caratterizzano la definizione di dipendenza esatta appena esposta. Tra queste la **dipendenza funzionale approssimata** (**AFD**) che rappresenta una dipendenza funzionale valida per quasi tutte le tuple di una relazione *r*. al fine di quantizzare l'errore di una AFD, sono state studiate diverse misure calcolabili in modo automatico.

I motivi per i quali si può preferire il calcolo di una dipendenza funzionale approssimata piuttosto che una dipendenza funzionale esatta sono molteplici. Fondamentalmente in applicazioni reali è difficile che i dati siano esenti da errori o da "sporcizia", di conseguenza un algoritmo che utilizza il calcolo di dipendenze funzionali esatte potrebbe portare alla luce pochi risultati o risultati banali. L'utilizzo di una adeguata soglia di errore nel calcolo di dipendenze funzionali approssimate potrebbe invece far emergere più dipendenze funzionali che possono essere considerate valide.

3.1.1 Dipendenze funzionali condizionate

Definizione: dati due sottoinsiemi di attributi X e Y di uno schema di relazione R, una **Dipendenza Funzionale Condizionata** (**CFD**) è una coppia $(X \rightarrow Y, S)$ dove S è un pattern di valori per gli attributi XY, denominato *pattern tableau* [35].

In contrasto con le dipendenze funzionali tradizionali o esatte (FD) che sono state sviluppate principalmente per la progettazione degli schemi, le dipendenze funzionali condizionate hanno lo scopo di catturare la consistenza dei dati incorporando il binding di valori nei dati semanticamente correlati. Fondamentalmente le dipendenze funzionali condizionate possono essere utilizzate in applicazioni di data cleaning [36].

3.2 Stato dell'arte

La modellazione data vault è stata appena esplorata nella letteratura accademica. Oltre alle specifiche del modello ufficiale [37], si conoscono solo un paio di opere: [38], che fornisce una concettualizzazione del modello fisico dei data vault, e [39], che descrive un approccio per la progettazione di data warehouse in cui la modellazione data vault è usata al posto degli schemi a stella o snowflake standard per implementare fisicamente il modello multidimensionale.

Il problema di come supportare o anche automatizzare la progettazione di data warehouse è stato ampiamente esplorato nella letteratura. Fondamentalmente, gli approcci alla progettazione data warehouse sono distinguibili in demand-driven e supply-driven [40]. Mentre il primo affronta il computo della modellazione multidimensionale basandosi principalmente sulle esigenze di business espresse dagli utenti, il secondo parte da un'analisi delle fonti dei dati.

I primi approcci alla progettazione supply-driven risalgono alla fine degli anni '90 [41] [42] [43] [44] [45] [46] [47] e propongono algoritmi che, assistiti da qualche interazione con il progettista, creano uno schema multidimensionale partendo dai diagrammi entity-relationship o dagli schemi relazionali. L'idea di base è quella di seguire le dipendenze funzionali (FD) espresse nello schema sorgente per costruire le gerarchie multidimensionali.

Negli anni seguenti, con l'avvento del così detto web warehouse, ci sono stati dei tentativi di ottenere degli schemi multidimensionali da sorgenti di dati in XML [48] [49] [50] [51]. In questo caso, il principale problema è che qualche dipendenza funzionale non è espressa intensionalmente, quindi è da controllare estensionalmente, interrogando propriamente il databse XML in fase di progettazione.

Più recentemente, alcuni lavori si sono concentrati su RDF analytics [52] [53] cioè su come interrogare i dati RDF in modo OLAP-like. Anche se sono ancora in una fase iniziale dello sviluppo, questi lavori possono aprire la strada verso la progettazione di schemi multidimensionali a partire da ontologie. Nella stessa direzione, in è proposto l'approccio

AMDO per derivare uno schema multidimensionale da una formalizzazione concettuale del dominio, in particolare, le sorgenti dati sono analizzate per ricercare potenziali pattern multidimensionali, successivamente i risultati sono combinati con la conoscenza espressa da un'ontologia di dominio per sostenere la ricerca dei requisiti.

3.3 Algoritmi

Gli algoritmi per l'estrazione delle dipendenze funzionali dai dati esistenti in letteratura si basano sui seguenti due tipi di approcci

- **top-down**: le dipendenze funzionali vengono generate per livelli usando un reticolo partendo da LHS piccoli, contenenti un solo attributo, ed accrescendoli attraverso l'aggiunta di attributi, così da eliminare il numero delle FD calcolate.
- **bottom-up:** le tuple vengono confrontate per avere agree-set o difference-set, dopodiché le FD candidate vengono dapprima generate e poi confrontate con gli agree-set/difference-set per testarne la validità

Gli algoritmi più studiati in letteratura sono: *TANE*, *FUN* e *FD_Mine* tutti basati su un approccio top-down. In particolare *TANE* e *FD_Mine* usano il metodo delle partizioni, mentre *FUN* utilizza il metodo dei free-set. Inoltre *FD_Mine* è l'algoritmo più veloce oggi disponibile in letteratura. Viceversa *Dep-Miner* e *Fast-FDs* si basano sull'approccio bottom-up [35].

3.3.1 TANE

Tane è stato il primo algoritmo proposto per l'estrazione delle dipendenze funzionali dai dati [54]. È molto efficiente, poiché sfrutta diverse tecniche di ottimizzazione, come l'uso delle partizioni e del pruning.

Metodo delle partizioni

Tane utilizza le *classi di equivalenza* degli insiemi, creando una partizione per ogni attributo *X* della relazione. Pertanto, la partizione *II(X)* conterrà tanti insiemi, quanti sono i diversi valori per l'attributo *X*

istanziati all'interno della relazione (vedi Figura 10). Ciascun insieme di una partizione conterrà, quindi, gli indici di riga delle tuple che condividono lo stesso valore per l'attributo *X*. Infine, gli insiemi di cardinalità 1 verranno eliminati dalle rispettive partizioni.

Per trovare una dipendenza funzionale $X \rightarrow Y$ basterà quindi confrontare le partizioni di *X* e *Y* per vedere se hanno gli stessi insiemi.

TupleID	A	B	C	D
1	1	a	$	Flower
2	1	A		Tulip
3	2	A	$	Daffodil
4	2	A	$	Flower
5	2	b		Lily
6	3	b	$	Orchid
7	3	C		Flower
8	3	C	#	Rose

Partitions of attributes

$\pi_{(A)} = \{\{1,2\},\{3,4,5\},\{6,7,8\}\}$
$\pi_{(B)} = \{\{1\},\{2,3,4\},\{5,6\},\{7,8\}\}$
$\pi_{(C)} = \{\{1,3,4,6\},\{2,5,7\},\{8\}\}$
$\pi_{(D)} = \{\{1,4,7\},\{2\},\{3\},\{5\},\{6\},\{8\}\}$

Figura 10 Partizioni TANE

Strategia di ricerca

Tane utilizza una strategia di ricerca top-down costruendo un reticolo (vedi Figura 11). Quest'ultima, per poter garantire la scoperta delle dipendenze funzionali minime, è composta da diversi livelli. In particolare un reticolo è un grafo diretto con il nodo radice (detto level-0) non avente alcun attributo.

I figli del nodo radice sono i nodi di livello 1, cioè quelli relativi alle partizioni sui singoli attributi e pertanto sono $\binom{m}{1}$ (dove *m* è il numero di attributi della relazione). Al livello 2, ogni nodo ha due attributi, per un totale di $\binom{m}{2}$ nodi e così via. L'ultimo livello avrà un solo nodo avente tutti gli attributi della relazione.

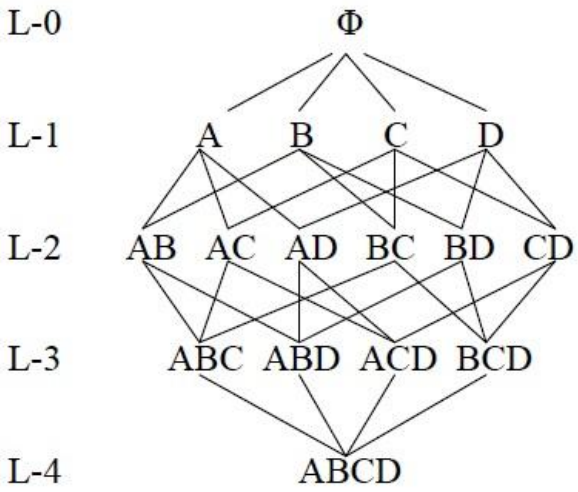


Figura 11 reticolo TANE

Esempio. Dato uno schema di relazione R con un insieme di attributi $T=\{A,B,C,D\}$ e un attributo $Y \in T$. Al primo livello si controlla per ogni attributo di $X=T-Y$, se $X \rightarrow Y$ è una dipendenza funzionale valida, nel qual caso si può effettuare il pruning di X da T.

Pruning

Il pruning è un'attività estremamente importante, dal momento che il numero di dipendenze funzionali può essere esponenziale sul numero di attributi. Tane effettua un pruning sull'albero delle ricerche ogni qual volta viene trovata una superchiave. In particolare se $X \rightarrow A$ è valida ed X è superchiave, allora si potrà escludere $X \cup A$ nella ricerca, quindi si taglieranno gli archi dal reticolo che vanno dal nodo X al livello successivo.

Dipendenze Approssimate

L'errore $e(X \rightarrow A)$ della dipendenza $X \rightarrow A$ è la minima frazione di tuple che devono essere rimosse dalla relazione affinché $X \rightarrow A$ sia valida. TANE ha una procedura che decide autonomamente il valore dell'errore.

Tempi di calcolo

Nel peggiore dei casi la ricerca è esponenziale rispetto al numero di attributi della relazione.

Criticità

Considerando $R = \{A, B, C\}$, con A e BC chiavi, a livello 1 Tane trova A chiave $\rightarrow B \in C^+(B)$ e $C \in C^+(C)$, per cui $A \rightarrow B$ e $A \rightarrow C$, e A viene eliminato dal reticolo.

Nel livello 2, Tane trova BC chiave → ma $C^+(AB)$ e $C^+(AC)$ non sono disponibili essendo stata eliminata A. In tal modo la dipendenza funzionale $BC \rightarrow A$ non verrebbe considerata.

Una eventuale soluzione potrebbe essere la modifica della regola di pruning, in modo che invece di eliminare A dal reticolo, viene marcata come non valida. I nodi non validi non sono più testati per la scoperta delle dipendenze funzionali, ma possono essere utilizzati per generare ulteriori nodi non validi fintanto che i loro set C^+ non sono vuoti [55].

3.3.2 FUN

Così come TANE, l'algoritmo FUN di Novelli e Cicchetti, attraversa il reticolo a livelli in maniera bottom-up e applica tecniche di raffinamento di partizione per trovare dipendenze funzionali [56] [57]. Tuttavia, FUN esplora un porzione più piccola dello spazio di ricerca attraverso una generazione dei candidati più restrittiva e un lazy look-up dei valori di cardinalità. La cardinalità di una combinazione dell'attributo X è il numero di valori distinti nella proiezione di X. Come la misura di errore in TANE, il valori di cardinalità possono essere utilizzati per ottimizzare la validazione delle dipendenze funzionali candidate.

Invece dei set C^+, FUN utilizza *free sets* e *non-free sets* per potare le FD candidate che producono solo FD non minimali. I free set sono set di attributi che non contengono nessun elemento che è funzionalmente dipendente su un subset del set rimanente. In altre parole, non esistono FD lungo gli attributi di un free set. L'insieme dei free set FS è definito come segue [56]:

- Sia $X \subseteq R$ un set di attributi in R, r sia una istanza di relazione di R e $|X|_r$ sia la cardinalità della proiezione di r su X, cioè il numero di valori distinti in X. Allora, $X \in FS_r \Leftrightarrow \nexists X' : |X'|_r = |X|_r$

Tutti i set di attributi che non sono in FS_r sono chiamati non-free set. Free set e non-free set implementano il pruning minimale, che è già conosciuto da TANE.

Solo i free set che hanno combinazioni non uniche di colonne sono considerati nella generazione dei candidati. La generazione anch'essa utilizza la funzione *apriori-gen*.

In qualche caso, tuttavia, l'algoritmo FUN è capace di dedurre la cardinalità delle combinazioni di attributi dai suoi subset. Questo è possibile se un set X è riconosciuto come non-free set. Allora, uno degli attributi del set X deve essere funzionalmente dipendente su uno dei subset diretti di X. Questo implica che uno dei subset di X ha la stessa cardinalità del set X. Questa regola può essere formalizzata come segue:

$$\forall X \notin FS_r, \forall X' \subset X : X' \in FS_r \Longrightarrow |X|_r = Max(|X'|_r) \quad (1)$$

La deduzione della cardinalità dei valori permette a FUN di fare un pruning sulle combinazioni di attributi più aggressivo di TANE: tutti i non-free set possono essere direttamente potati dal reticolo, perché se è necessaria la cardinalità di un superset dopo, l'algoritmo può dedurre questa cardinalità dai sottoinsiemi dei superset. TANE, dall'altro lato, ha bisogno di processare questi insiemi di candidati più volte, finche il loro C^+ diventa vuoto. Pertanto, la deduzione della cardinalità porta al maggior vantaggio prestazionale di FUN.

Esempio

Il seguente esempio illustrato in Figura 12 spiega il processo di pruning e deduzione in maniera più dettagliata.

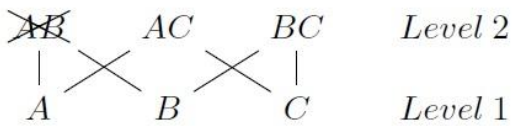


Figura 12 Esempio di reticolo potato in FUN

Considerando un dataset $R = \{A, B, C\}$ e le minime dipendenze funzionali $A \rightarrow B$ e $BC \rightarrow A$. A livello L_1, FUN scopre la dipendenza funzionale $A \rightarrow B$ e pota il nodo AB dal reticolo nel livello L_2 perché è un non-free set dovuto alla FD trovata. Per il livello L_2, la generazione di

candidati di FUN omette il nodo *ABC* di livello L_3 in quanto è un subset di *AB* che è già stato potato e non c'è quindi un candidato per la parte sinistra di una dipendenza funzionale. Per trovare la dipendenza funzionale $BC \rightarrow A$, FUN ha ancora bisogno di confrontare la cardinalità di *BC* e *ABC*. Poiché il nodo *ABC* è stato potato, deve essere un non-free set e la sua cardinalità può essere dedotta dai suoi subset diretti seguendo l'equazione 1 [58].

3.3.3 FD_mine

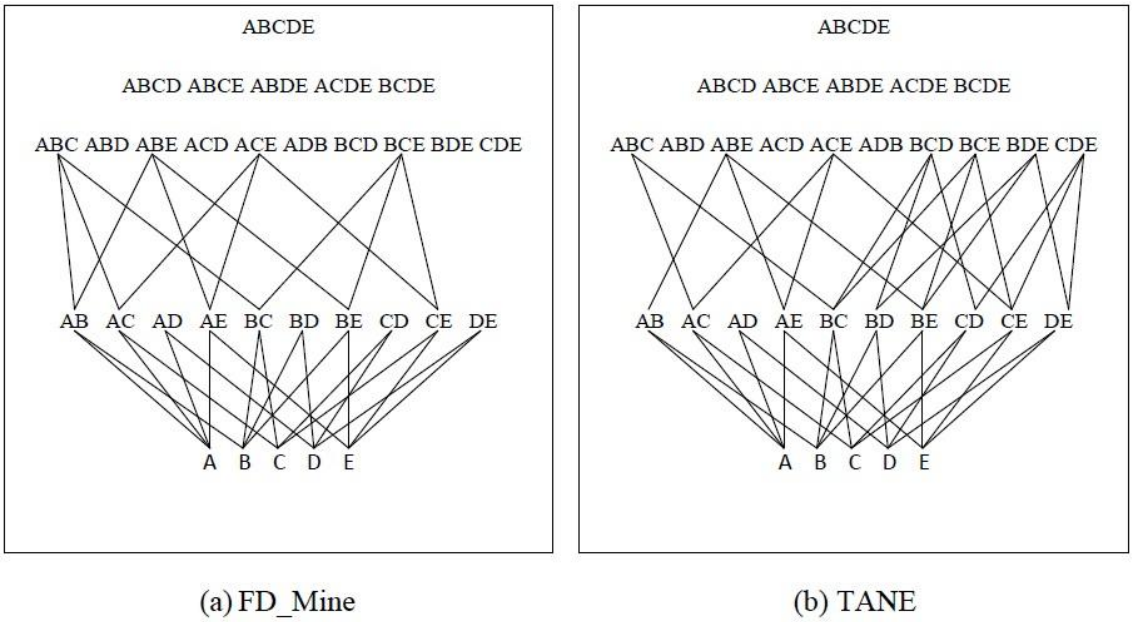
FD_mine [59] può essere visto come un'evoluzione di Tane. Esso sfrutta la proprietà di simmetria delle dipendenze funzionali:

- Se $X \rightarrow Y$ e $Y \rightarrow X$ allora $X \leftrightarrow Y$, cioè i due attributi sono equivalenti

Così che, sfruttando gli Assiomi di Armstrong, si ha che:

- Dato $X \leftrightarrow Y$, se vale $XW \rightarrow Z$, allora vale $YW \rightarrow Z$
- Dato $X \leftrightarrow Y$, se vale $ZW \rightarrow X$, allora vale $ZW \rightarrow Y$

Infine, FD_Mine usa anche l'approccio a livelli, in modo da ridurre drasticamente il numero di attributi su cui cercare (si veda Figura 13).



(a) FD_Mine (b) TANE

Figura 13 Confronto tra TANE e FD_Mine

Criticità

Si dimostra che FD_Mine può condurre alla scoperta di dipendenze funzionali non minime. Assumendo X equivalente a Y e $XW \rightarrow Z$ (in rispetto dell'assioma $YW \rightarrow Z$) dato un attributo A di Y dipendente da W', subset di W allora $YW \rightarrow Z$, ma un subset di YW, chiamato $YW - A$, determina già Z di conseguenza la dipendenza funzionale non è minima [55].

3.4 Implementazione distribuita su Apache Spark

L'approccio che abbiamo adottato per determinare le dipendenze funzionali approssimate (AFDs) è un adattamento dell'algoritmo TANE.

Data una tabella r con schema R, TANE computa tutte le AFDs valide $X \rightsquigarrow a$ con X⊆R e $a \in R \backslash X$ basandosi su una strategia di enumerazione basata su livelli per navigare lo spazio di ricerca di tutti i possibili subset di R (cioè, il reticolo di contenimento).

Anche se TANE applica una serie di regole di pruning per evitare il calcolo/ritorno di dipendenze banali e non minime, la sua complessità rimane esponenziale a causa del numero di attributi candidato dell'insieme X. Nello specifico, il caso peggiore di complessità di TANE è $O(|r| + |R|^{2.5})2^{|R|}$, dove $|r|$ è la cardinalità della tabella r e $|R|$ è il numero di attributi. Si può notare che, nel nostro caso, possiamo restringere la nostra ricerca a semplici AFDs ($|X| = 1$). Verrà ora spiegata una strategia di enumerazione che lavora con AFDs semplici e riduce la complessità di TANE a $O(|r| \cdot |R^2|)$ nel caso peggiore e a $O(|r| \cdot |R|)$ nel caso migliore.

Partiamo considerando le Dipendenze Funzionali "tradizionali" (FDs). Dato uno schema R, l'insieme delle FDs candidate $a \rightarrow b$ con $a, b \in R$ può essere rappresentato usando una matrice $|R| \times |R|$ che chiameremo Z di cui le righe e le colonne rappresentano rispettivamente la parte sinistra e la parte destra di una dipendenza funzionale. In modo che $Z[a, b]$ corrisponda a $a \rightarrow b$. Se la FD $a \rightarrow b$ è determinata nei dati memorizzati allora la cella $Z[a, b]$ è settata a *true*, altrimenti è settata a *false*.

Un approccio per riempire Z è quello di controllare ogni singola cella, cioè, calcolare qualsiasi FD possibile. In realtà, molti controlli possono essere evitati da un esplorazione ordinata delle celle di Z. la strategia di esplorazione richiede che le righe e le colonne di Z debbano essere ordinare per cardinalità decrescente del rispettivo dominio dell'attributo che rappresentano.

Data la matrice ordinata, si può notare che solo le celle sopra la diagonale potranno essere *true*, tutte le altre saranno per forza *false* in quanto:

- le celle lungo la diagonale corrispondono a dipendenze funzionali banali come $a \rightarrow a$.
- le celle al di sotto della diagonale invece corrispondono a dipendenze funzionali irrealizzabili come $b \rightarrow a$ con $|b| < |a|$.

Per quanto riguarda le celle al di sopra della diagonale invece, possiamo evitare quei check che corrispondono a Dipendenze Funzionali transitive applicando la seguente strategia di esplorazione:

- *Regola 1*: Fare il check delle celle $Z[a, b]$ dove $|b|$ è massimo e, lungo queste, dare la priorità alla cella con $|a|$ minimo.
- *Regola 2*: Se la dipendenza funzionale corrispondente alla cella $Z[b, c]$ è stata settata a *true*, allora si possono settare a *true* tutte le dipendenze funzionali corrispondenti alle celle $Z[*, c]$ tali che sia $Z[*, b]$

Per capire perché le regole 1 e 2 evitano il check di dipendenze funzionali transitive proviamo a considerare le dipendenze funzionali $a \rightarrow b$ e $b \rightarrow c$, che transitivamente implicano $a \rightarrow c$. Allora questo significa che $|c| \leq |b| \leq |a|$, quindi seguendo la regola 1 il check di $a \rightarrow c$ dovrà essere fatto dopo il check di $a \rightarrow b$ e il check di $b \rightarrow c$. Ma dato che è vero $b \rightarrow c$ la regola 2 ci permette di settare a *true* $a \rightarrow c$ prima di farne il check.

In accordo con la precedente regola di enumerazione, il numero di dipendenze funzionali candidate che deve essere verificato dipende, dato il numero di attributi, dal numero di dipendenze funzionali transitive in R. Il caso peggiore si verifica quando non ci sono dipendenze funzionali

transitive tra attributi in R, dato che tutte le celle nella metà superiore di Z, cioè $|R| \times (|R| - 1)/2$, dovranno essere controllate. Il caso migliore invece si verifica quando gli attributi di R sono coinvolti in una gerarchia di tipo lineare, dato che il numero di check scende a $|R| - 1$. Considerando che la complessità di TANE è determinata dalla sua strategia di enumerazione e che TANE controlla le dipendenze funzionali in un tempo lineare, la complessità di questo approccio risulta essere $O(|r| \cdot |R|^2)$ nel caso peggiore e $O(|r| \cdot |R|)$ nel caso migliore.

3.4.1 Calcolo di Dipendenze Funzionali Approssimate (AFDs)

La strategia di enumerazione descritta per le dipendenze funzionali tradizionali si basa sull'ordinamento degli attributi. Sfortunatamente, quando si lavora con le dipendenze funzionali approssimate (AFDs) bisogna permettere qualche tolleranza sulla cardinalità degli attributi (quindi anche sull'ordinamento degli attributi) per permettere possibili errori nei dati.

Consideriamo due attributi a e b tali che $|a| \gtrsim |b|$. Se si volessero cercare solo dipendenze funzionali tradizionali bisognerebbe calcolare $a \rightarrow b$ e non $b \rightarrow a$ ($Z[b, a]$ si trova nella parte inferiore della matrice e sarebbe da saltare) ma in realtà, considerando le dipendenze funzionali approssimate, bisogna anche considerare che l'alta cardinalità di a sia dovuta a errori nei dati. In altre parole, bisogna controllare anche $b \rightsquigarrow a$.

In pratica questa situazione si presenta se $|a| - \epsilon < |b| < |a|$. Quindi, per preservare la correttezza della strategia di enumerazione, quando si tratta con le dipendenze funzionali approssimate bisogna controllare entrambe le celle $Z[a, b]$ e $Z[b, a]$ quando $abs(|a| - |b|) < \epsilon$. Ovviamente, come side effect, la capacità di pruning sarà ridotta dal fatto che più celle avranno bisogno di essere controllate.

Tuttavia, la complessità del caso migliore e del caso peggiore rimarranno senza cambiamenti.

Implementazione su Spark

Nel lavoro di tesi il controllo delle dipendenze funzionali è stato fatto attraverso l'implementazione dell'algoritmo appena descritto e attraverso

delle query SQL che avevano il compito di calcolare il grado di correlazione tra attributi.

L'implementazione della matrice è stata svolta estraendo per ogni attributo il valore di count e di count distinct, dopodiché tutti gli attributi sono stati ordinati per count distinct. In questo modo è stato possibile costruire la matrice ordinata.

Successivamente si è poi passati al controllo delle dipendenze funzionali tra i vari attributi. Ad esempio, il controllo della dipendenza funzionale $Z[a, b]$ (dove $|a| > |b|$) viene fatto con le seguenti query:

- Questa query restituisce un valore, chiamato *sumValid* identifichiamo con pq_clinical la tabella nella quale sono inseriti tutti i dati clinici:

```
SELECT sum(valid)
FROM(
     SELECT max(groupCount) valid
     FROM(
          SELECT a,b,count(*) groupCount
          FROM pq_clinical
          WHERE a is not null and b is not
null
          GROUP BY a,b
     )groupCounts
     GROUP BY a
) validCounts;
```

- Attraverso questa query invece otteniamo un valore che chiameremo *groupCard*:

```
SELECT count(*)
FROM pq_clinical
WHERE a is not null and b is not null;
```

Una volta ottenuti i valori di sumValid e groupCard l'approssimazione è definita come:

$$error = 1 - sumValid/groupCard$$

Questo valore, compreso tra 0 e 1, rappresenta l'errore di correlazione tra i 2 attributi, quindi quando è 0 ci troviamo in un caso di dipendenza funzionale esatta (FD).

Come già detto in precedenza, per il calcolo delle dipendenze funzionali approssimate (AFD) bisogna fare inoltre un altro tipo di controllo in quanto c'è la possibilità che il rumore dei dati abbia alterato la cardinalità degli attributi.

Bisogna quindi calcolare, attraverso un'altra query SQL, due valori che chiameremo *cardA* e *cardB*:

```
SELECT  count(distinct a) as cardA,
        count(distinct b) as cardB
FROM pq_clinical
WHERE a is not null and b is not null;
```

Una volta calcolati i valori, se si verifica la condizione:

$$|cardA - cardB| < Threshold * groupCard$$

dove $Threshold$ è l'approssimazione massima accettata, bisognerà fare anche il controllo di $Z[b, a]$.

Esempio

La Figura 14 evidenzia 3 differenti tipi di gerarchia che, dall'alto al basso, rappresentano il caso migliore, intermedio e peggiore per la strategia di enumerazione adottata.

Nel caso migliore solo 3 dipendenze funzionali sono state calcolate perché tutte le altre sono transitive e dedotte utilizzando la regola 2.

Nel caso intermedio 5 dipendenze funzionali sono calcolate. $Z[a, c]$ deve essere calcolata in questo caso in quanto non può essere derivata transitivamente ($Z[b, c]$ è *false*).

Infine, nel caso peggiore, tutte le sei celle dovranno essere calcolate.

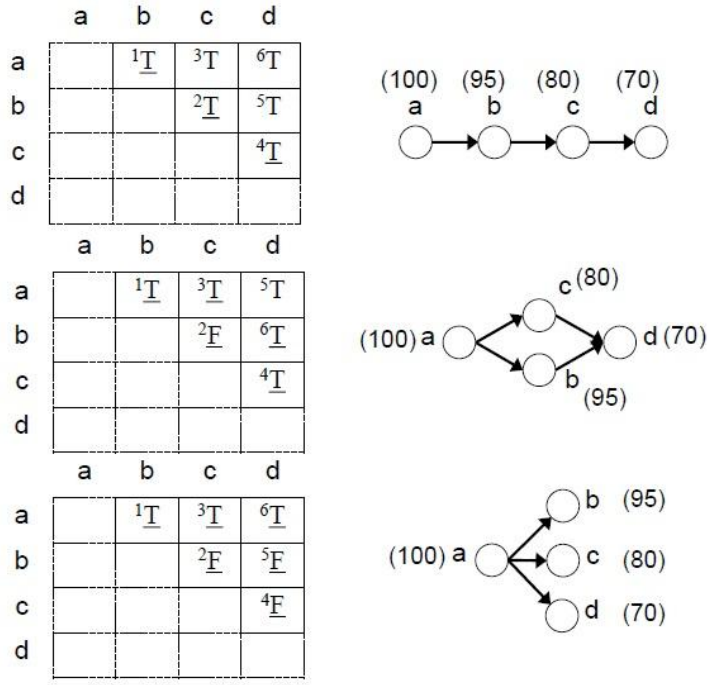


Figura 14 Le matrici per 3 gerarchia. I valori vicini agli attributi denotano la loro cardinalità, i numeri nelle celle denotano l'ordine di enumerazione, le FD a cui è stato fatto il check sono sottolineate

4 Modellazione multidimensionale del Genome Space

4.1 Scoperta di gerarchie dal database TCGA

All'interno di questa sezione verranno presentate tutte le varie fasi di cui è composto il lavoro di tesi, partendo dall'estrazione dei dati dal database TCGA fino alla presentazione dei risultati ottenuti.

4.1.1 Metodologia

Estrazione dei dati

Come spiegato nel capitolo 2 le modalità di estrazione dei dati dal portale TCGA sono principalmente 4: Data Matrix, Bulk Download, Access Http Directory e File Search. Attraverso l'applicativo File Search è stato possibile limitare e raffinare la ricerca di determinati tipi di dato e di avere una stima preliminare della quantità di file e di dati da scaricare.

Si è quindi deciso di utilizzare l'applicativo File Search e di concentrarsi sui dati clinici presenti all'interno del portale. Tutti i dati clinici presenti all'interno del database sono di data level 1 e 2. Mentre i dati di livello 1 sono memorizzati su diversi formati come file xml, file pdf e file txt i dati di livello 2 invece sono tutti unicamente memorizzati in più file txt.

Considerando inoltre che i dati di livello due, per definizione, sono l'aggregazione e la sintesi dei dati di livello 1, si è deciso di concentrarsi solo su tali dati. In questo modo si è riusciti a ridurre la mole di dati e il lavoro riguardante l'estrazione e il caricamento delle informazioni senza perdere contenuto informativo.

Caricamento e denormalizzazione

Una volta ottenuti tutti i file riguardanti i dati clinici di secondo livello attraverso un applicativo Java si è passati al caricamento degli stessi in uno schema riconciliato. Si sono quindi analizzati tutti i file ricercando tutti gli attributi, una volta ottenuti tutti gli attributi (1016) si è creata una tabella Hive per la memorizzazione.

Successivamente ogni file è stato analizzato e caricato nello schema riconciliato creato inserendo valori NULL nei campi dove non erano presenti dati.

Il numero totale delle righe presenti nello schema riconciliato finale è di 452122.

Infine si è poi deciso di trasferire tutti i dati dalla tabella hive ad una nuova tabella *parquet* in quanto, dovendo fare analisi riguardanti singoli attributi, si è ritenuto un database di tipo colonnare più adatto e soprattutto più performante per questo tipo di analisi.

Algoritmo per le Dipendenze Funzionali

Dopo aver effettuato il lavoro di estrazione e caricamento dei dati in uno schema riconciliato e denormalizzato è stato possibile lanciare l'algoritmo per la scoperta delle dipendenze funzionali.

La nostra matrice è quindi una matrice quadrata 1016*1016 e il numero potenziale massimo di check che l'algoritmo dovrà eseguire è di:

$$(1015 * 1016)/2 = 515620$$

Mentre il numero minimo è di 1016.

La durata totale dell'algoritmo è stata di circa 68 ore ed i risultati sono stati i seguenti:

Algoritmo su tutto TCGA	
Dipendenze Funzionali calcolate	1583
Dipendenze Funzionali transitive	241273
Dipendenze Funzionali totali	242856

Filtraggio degli attributi

A fronte di un elevato numero di risultati, si è deciso di limitare l'ampiezza dell'algoritmo in modo da poter avere un numero di dati facilmente visualizzabili, anche graficamente, e poter dare valore ai risultati ottenuti.

Si è quindi deciso di isolare solamente un determinato numero di attributi, ci si è concentrati quindi sugli attributi più popolati, in modo da aver dipendenze funzionali effettivamente sostenute da una buona quantità di valori. Come attributi sono quindi stati scelti i 150 più popolati.

Nel caso di 150 attributi, il numero di check che l'algoritmo deve controllare è di:

$$(149 * 150)/2 = 11175$$

Mentre l'algoritmo su tutto il database è stato lanciato con un parametro di approssimazione del 5%, l'algoritmo sui primi 150 attributi è stato lanciato prima con una soglia del 5% e poi del 1%, di seguito i risultati:

Soglia del 5%	
Dipendenze Funzionali calcolate	277
Dipendenze Funzionali transitive	2241
Dipendenze Funzionali totali	2518

Soglia dell'1%	
Dipendenze Funzionali calcolate	247
Dipendenze Funzionali transitive	1866
Dipendenze Funzionali totali	2113

Si può notare che attraverso l'abbassamento della soglia dal 5% all'1% non sono state più rilevate 30 dipendenze funzionali dirette (1,08% sul totale), le quali servivano per trovare altre 375 dipendenze funzionali transitive.

Divisione per malattia

Una volta eseguito l'algoritmo per i primi 150 attributi più popolati si è deciso di focalizzarsi su un ristretto numero di malattie, si sono quindi presi in esame 3 diversi tipi di tumore:

- Carcinoma invasivo del seno (Breast invasive carcinoma - BRCA)
- Adenocarcinoma del Colon (Colon adenocarcinoma - COAD)

- Carcinomi squamocellulari del distretto testa-collo (Head and Neck squamous cell carcinoma)

Anche in questo caso, sono stati isolati, per ogni tipo di tumore, i 100 attributi più popolati. Oltre alla selezione dei 100 attributi più popolati, seguendo le indicazioni di esperti del settore, è stato fatto un ulteriore filtraggio di attributi, eliminando quelli considerati poco utili.

Il risultato finale di questo filtraggio è riassunto nella tabella seguente:

Tumore	**N. Attributi**
Breast invasive carcinoma	86
Colon Adenocarcinoma	76
Head and Neck squamous cell carcinoma	80

L'algoritmo di scoperta di dipendenze funzionali è quindi stato eseguito 3 volte, una per ogni tipo di tumore, con una soglia dell'1% per fare in modo di limitare il più possibile il numero dei risultati.

Per quanto riguarda BRCA, il numero di check che l'algoritmo deve controllare è di:

$$(85 * 86)/2 = 3655$$

I risultati sono:

Soglia del 1%	
Dipendenze Funzionali calcolate	142
Dipendenze Funzionali transitive	801
Dipendenze Funzionali totali	943

Per quanto riguarda COAD, il numero di check che l'algoritmo deve controllare è di:

$$(75 * 76)/2 = 2850$$

I risultati sono:

Soglia del 1%	
Dipendenze Funzionali calcolate	121
Dipendenze Funzionali transitive	878

Dipendenze Funzionali totali	999

Per quanto riguarda HNSC, il numero di check che l'algoritmo deve controllare è di:

$$(79 * 80)/2 = 3160$$

I risultati sono:

Soglia del 1%	
Dipendenze Funzionali calcolate	123
Dipendenze Funzionali transitive	1269
Dipendenze Funzionali totali	1392

4.1.2 Risultati

I risultati di queste 3 elaborazioni vengono visualizzati in maniera abbastanza efficace attraverso un grafo aciclico diretto (Directed Acycling Graph - DAG) in cui vengono visualizzate soltanto le dipendenze funzionali dirette, cioè quelle calcolate, e non quelle transitive in quanto non aggiungono contenuto informativo e minano la leggibilità del grafo.

In Figura 15 è possibile vedere il risultato dell'algoritmo per quanto riguarda il carcinoma invasivo del seno (BRCA).

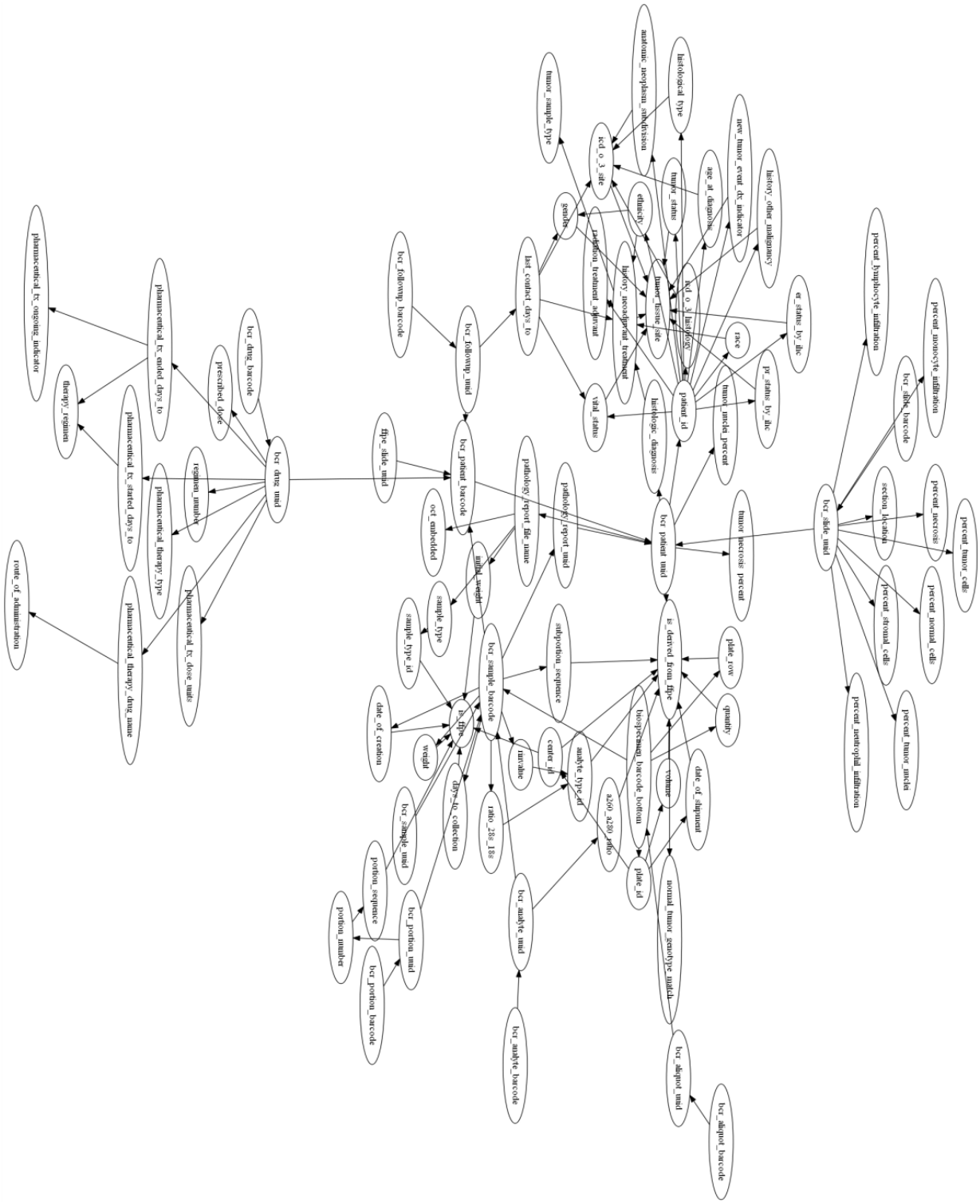


Figura 15 Grafo Dipendenze Funzionali BRCA

Per poter semplificare ancora di più la rappresentazione è stato fatto del pruning del grafo eliminando l'attributo *tumor_type*, il quale è stato creato ad hoc durante la fase di caricamento con lo scopo di identificare la provenienza dei dati in base al tumore. Dopo il pruning il numero di dipendenze funzionali dirette è sceso da 142 a 119. Il numero di righe

nel database riconciliato che si riferivano a questo tipo di tumore erano 46174.

In Figura 16 invece è possibile vedere il grafo delle dipendenze funzionali per quanto riguarda l'Adenocarcinoma del Colon, anche in questo caso è stato fatto del pruning sull'attributo *tumor_type* e le dipendenze funzionali dirette sono passate da 121 a 87. Il numero di righe nel database riconciliato che si riferivano a questo tipo di tumore erano 22734.

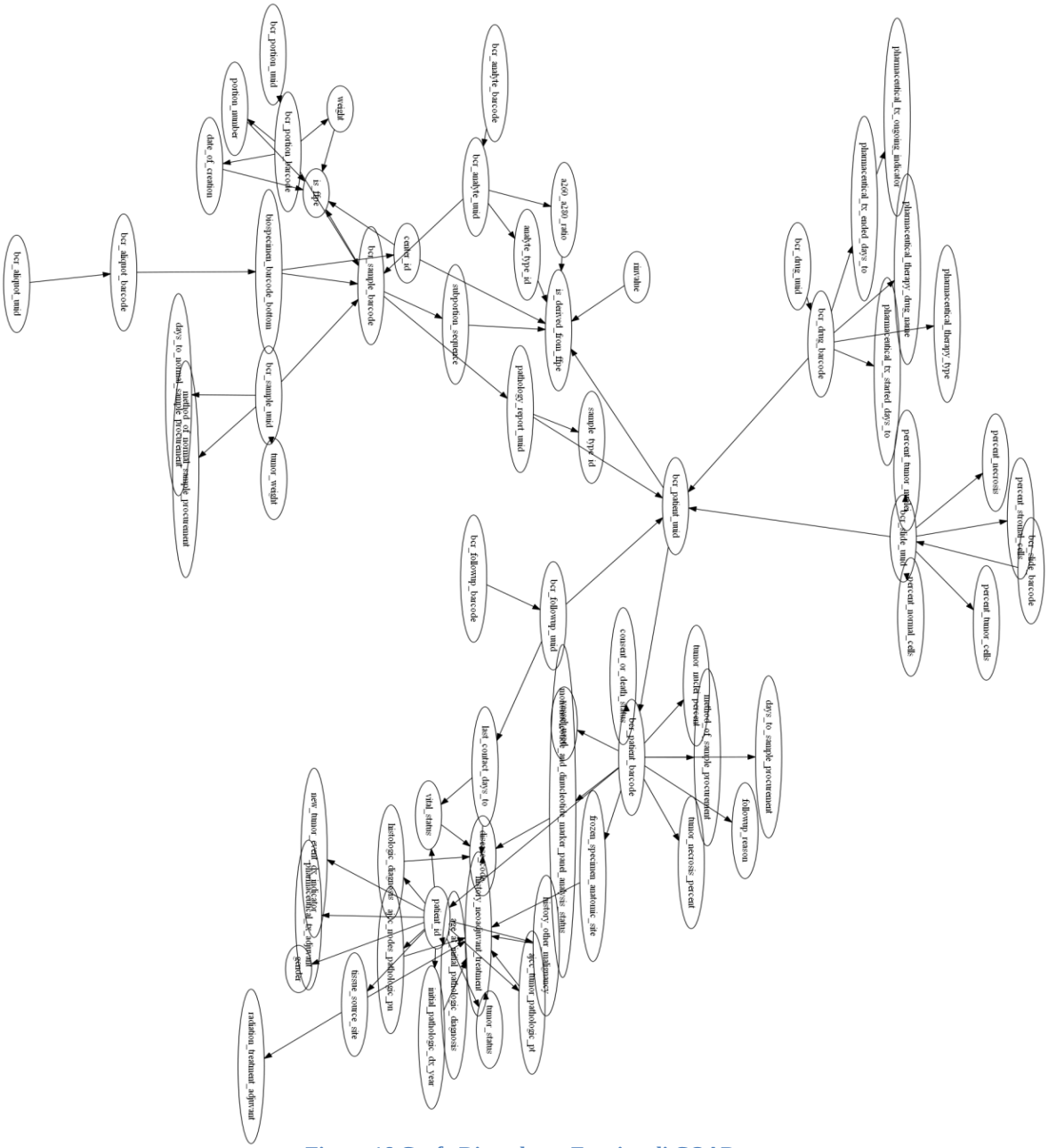


Figura 16 Grafo Dipendenze Funzionali COAD

L'ultimo grafo (vedi Figura 17) rappresenta il grafo delle dipendenze funzionali dirette relative ai carcinomi squamocellulari del distretto testa-collo. Anche in questo caso, dopo il pruning sull'attributo tumor_type le dipendenze funzionali dirette sono passate da 123 a 82. Il numero di righe nel database riconciliato che si riferivano a questo tipo di tumore erano 20080.

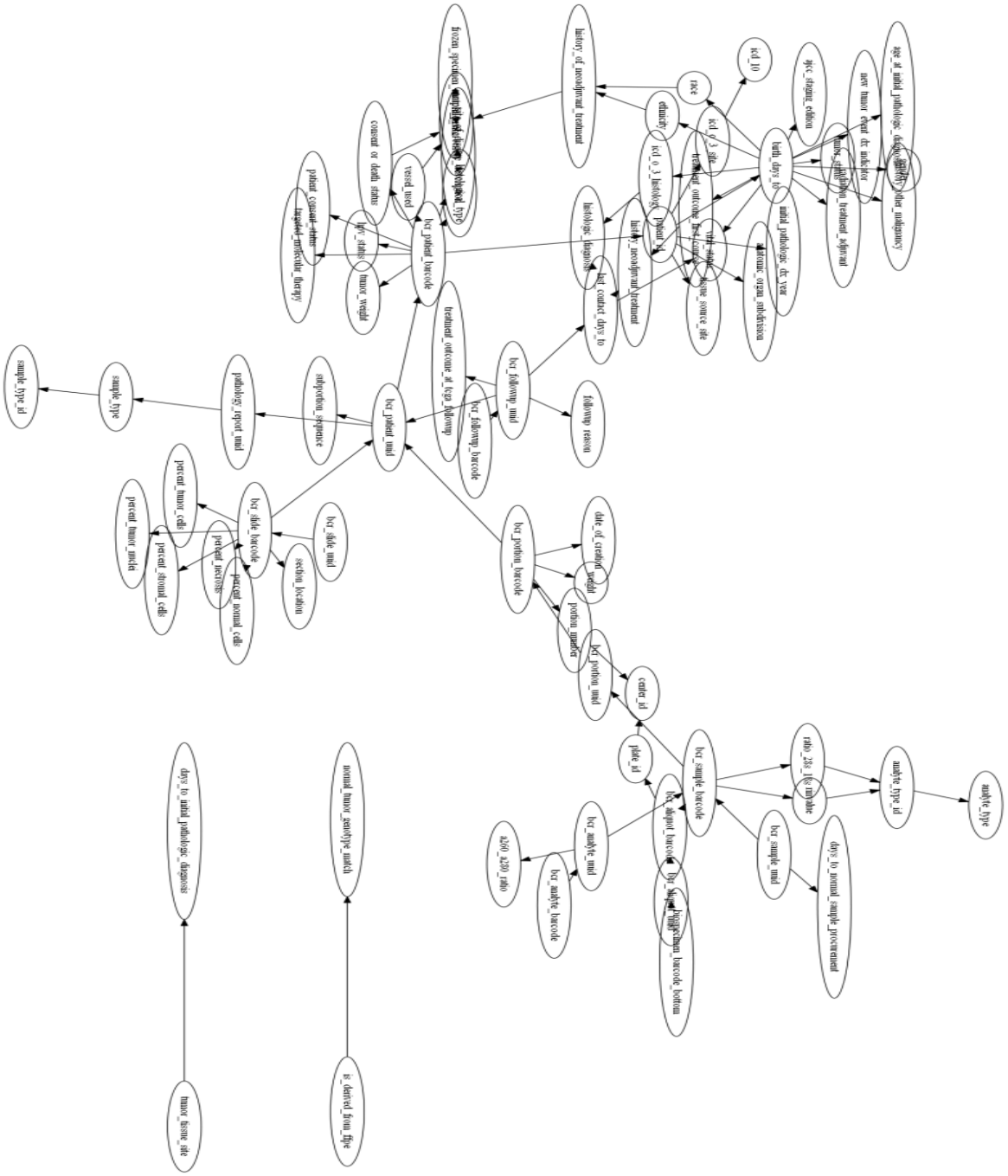


Figura 17 Grafo Dipendenze Funzionali HNSC

Il numero di istanze per ogni singola malattia influisce notevolmente sul numero di dipendenze che si possono trovare, infatti le istanze relative a BRCA sono più del doppio rispetto alle istanze COAD e HNSC ed infatti le dipendenze funzionali dirette passano da 82 e 87 di COAD e HNSC a 119 per BRCA.

4.2 Il cubo Mapping

Le sessioni di analisi in biologia sono intrinsecamente esplorative e dinamiche, quindi un approccio OLAP è adatto in quanto fornisce strumenti potenti e flessibili per esplorare i dati provenienti da diverse prospettive e diversi livelli di dettaglio. Quindi si possono utilizzare i classici operatori OLAP come roll-up, slice-and-dice etc. Ovviamente, un ruolo primario nel garantire l'efficacia di analisi OLAP è svolto dal modello multidimensionale adottato per rappresentare i dati.

Il cubo Mapping basato sulla notazione DFM [60], è visualizzabile in Figura 18. Ogni Mapping è associato ad una regione di un campione di riferimento ed un campione di input, ed è descritto da una serie di misure statistiche: *count*, *media*, *max*, *min* e *sum*. In particolare, ad eccezione del *count*, le altre misure sono calcolate su una caratteristica delle specifiche regioni presenti nel campione di input. Dal momento che tali caratteristiche non sono note a priori, viene usata la dimensione *feature* per supportare un numero variabile di misure. Per esempio, la misura *Max* di un mapping associato con la feature *p-value* si riferisce al massimo p-value trovato nella regione mappata. Come denotato dal cerchio con doppio bordo in Figura 18 reference e input condividono la stessa gerarchia. La gerarchia *Sample* riflette principalmente l'ENCODE data model [61]. Un *experiment* è il risultato di una singola analisi biologica su una porzione, chiamata *cell line*, di un *tissue* (tessuto) estratto da un paziente. In particolare, il significato di un esperimento dipende dal suo specifico tipo (Chip-Seq, DNA-Seq etc.). I campioni, che sono l'input necessario per una operazione di Map, sono il risultato di una o più operazioni GMQL su di un set di esperimenti.

Ora bisognerà collegare le gerarchie ottenute dagli algoritmi riguardanti i 3 tipi di tumori analizzati. Il punto di contatto è l'attributo *Cell Line* al

quale andrà collegato il nostro identificativo del paziente. Tutti gli altri attributi non raggiungibili dal paziente verranno potati.

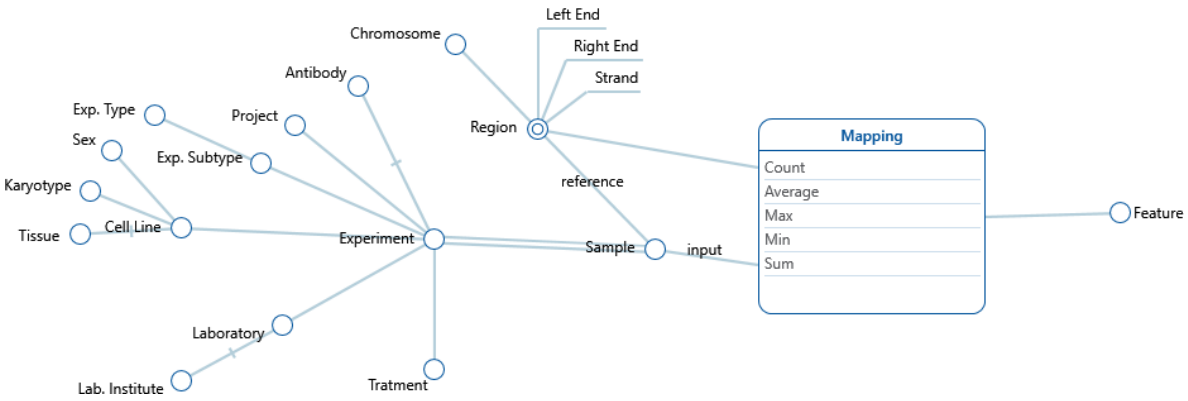


Figura 18 Mapping Cube originale

Possiamo quindi vedere come la gerarchia trovata eseguendo l'algoritmo per il tipo di tumore BRCA si può inserire all'interno dello schema sopra riportato. In Figura 19 il risultato iniziale.

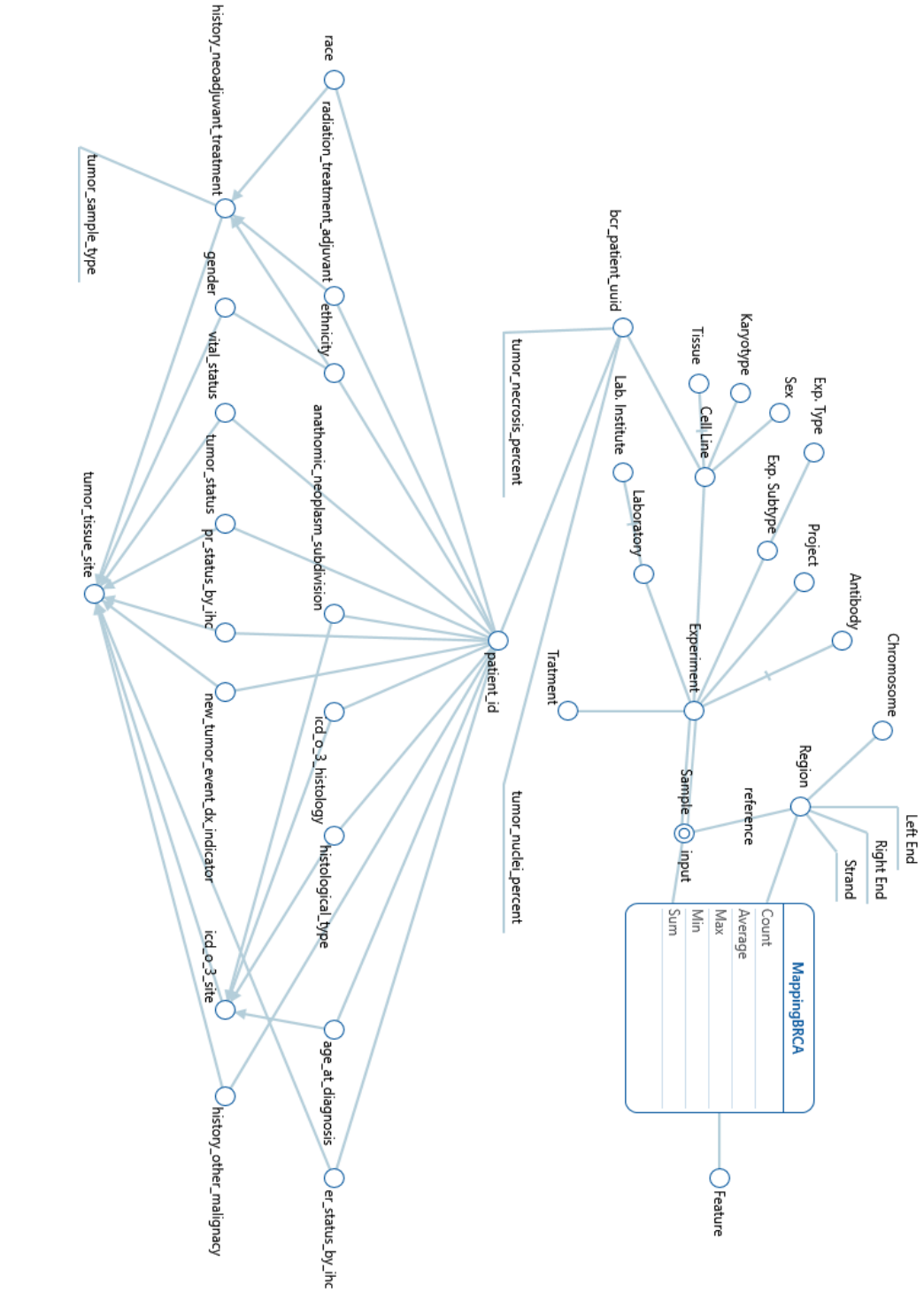


Figura 19 Mapping Cube con gerarchia BRCA non potata

Come si può notare dal DFM (Dimensional Fact Model) in figura su gran parte della gerarchia trovata dall'algoritmo per quanto riguarda i tumori di tipo BRCA è stato fatto del pruning, infatti tutti gli attributi non raggiungibili dall'attributo identificativo del paziente *bcr_patient_uuid* non è stato possibile inserirli.

La fase di pruning può continuare in quanto si può notare che gli attributi *bcr_patient_uuid* e *patient_id* sono due identificatori univoci del paziente, questo significa che avranno una corrispondenza 1:1, si decide quindi di eliminare dalla gerarchia trovata l'attributo *patient_id*. Il risultato è in Figura 20.

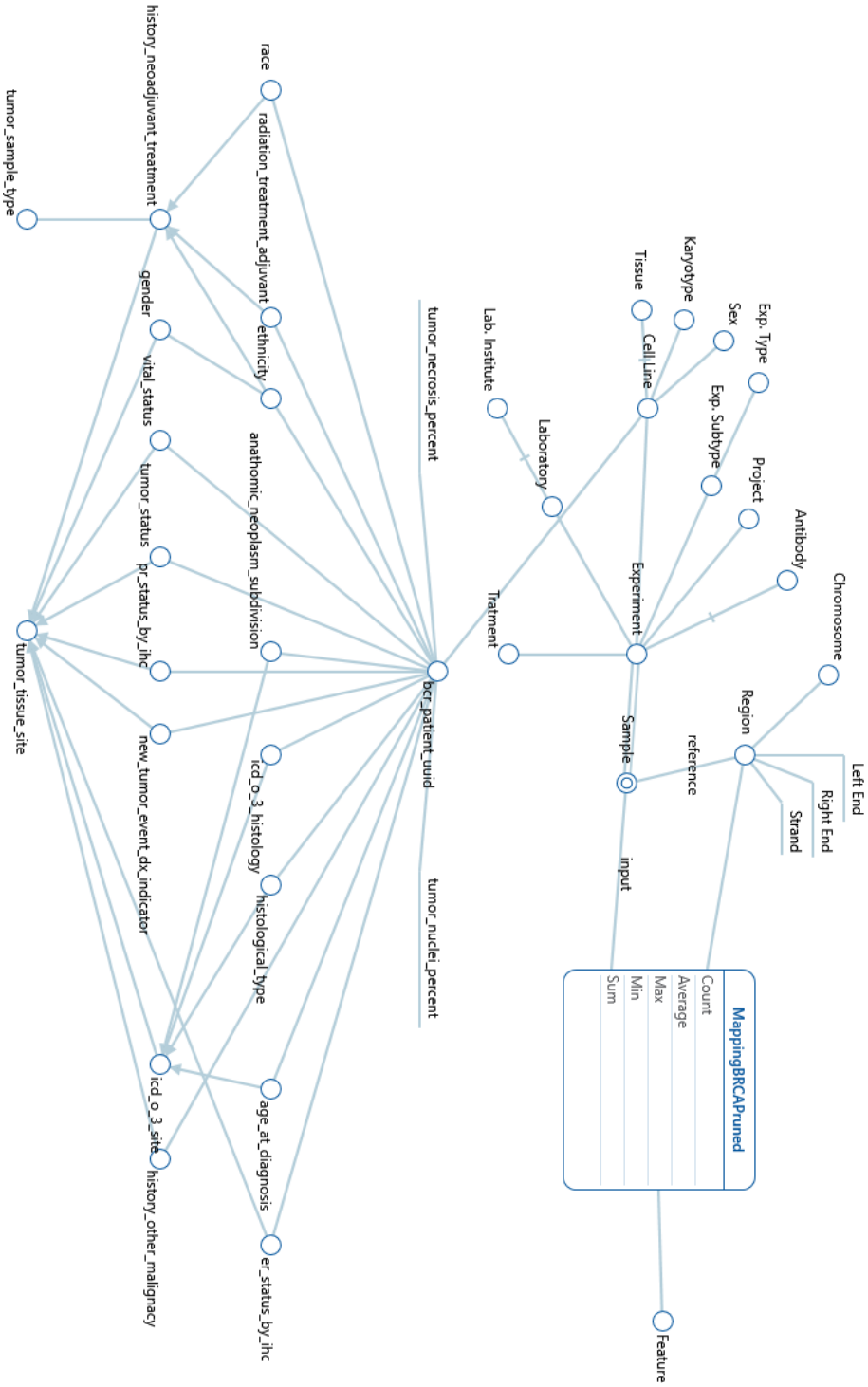


Figura 20 Mapping Cube con gerarchia BRCA pruned

La stessa operazione è possibile farla anche per la gerarchia trovata sui tumori COAD, in Figura 21 il risultato iniziale senza pruning degli attributi con corrispondenza 1:1.

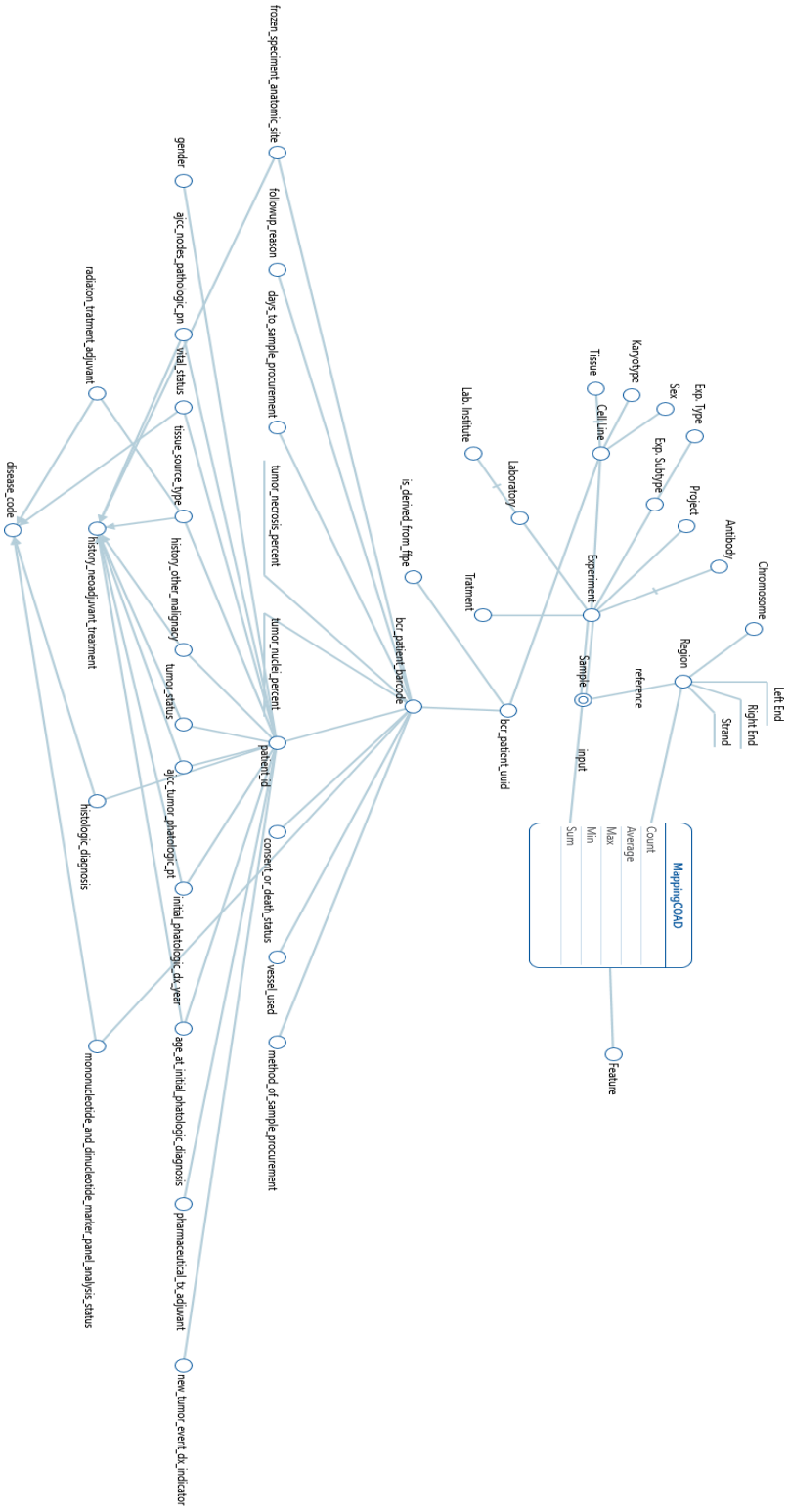


Figura 21 Mapping Cube con gerarchia COAD non potata

Anche in questo caso ci sono i due attributi *bcr_patient_uuid* e *patient_id*, si decide quindi di eseguire la stessa potatura come nel caso BRCA, il risultato finale in Figura 22.

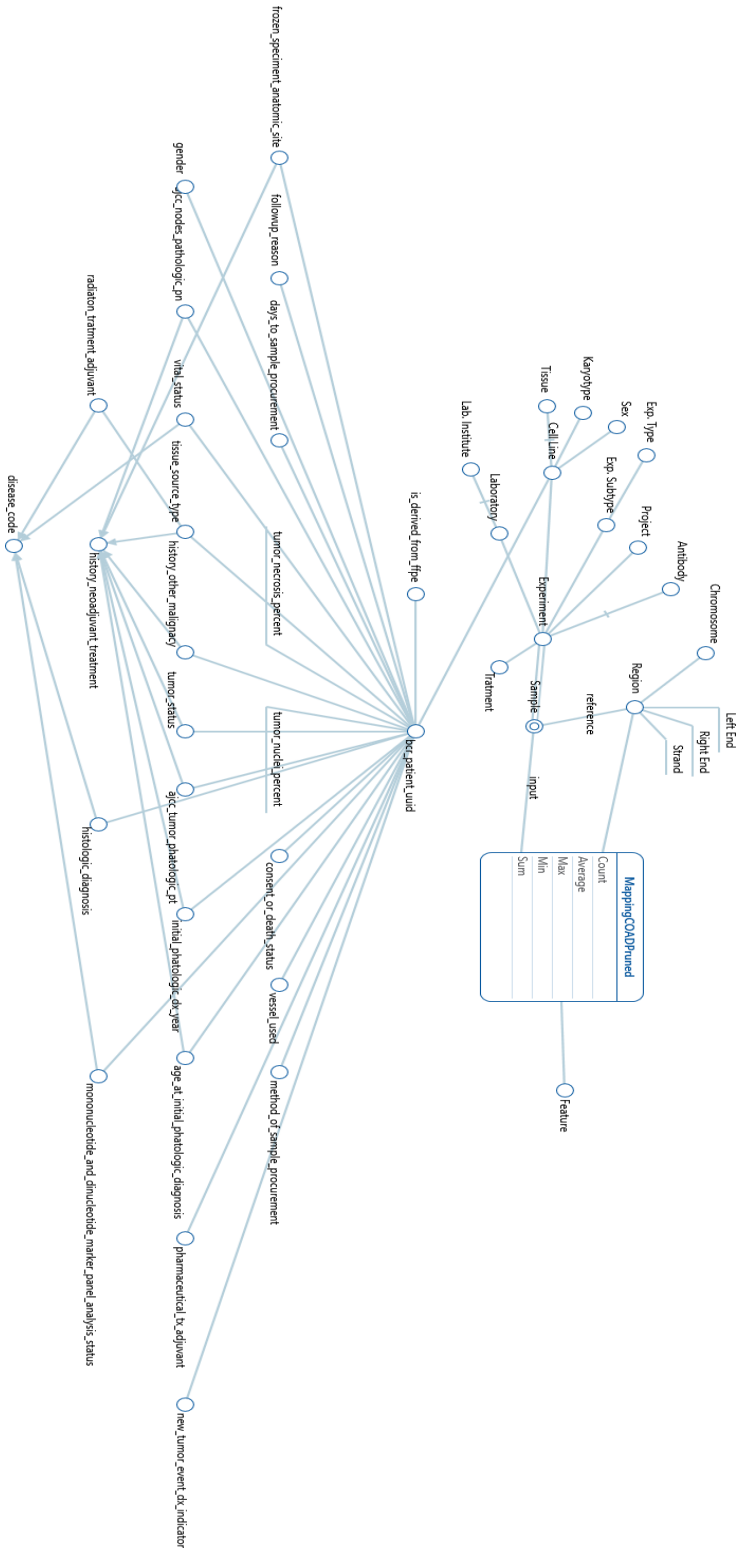


Figura 22 Mapping Cube con gerarchia COAD pruned

Infine, anche per il tumor type HNSC viene svolta la stessa operazione, viene quindi inserita la gerarchia del paziente nell'attributo *cell line*, in Figura 23 il risultato.

Figura 23 Mapping Cube con gerarchia HNSC non potata

Anche in questo caso, la corrispondenza tra *bcr_patient_uuid* e *patient_id* viene riscontrata, viene quindi eliminato il campo *patient_id*, il risultato in Figura 24.

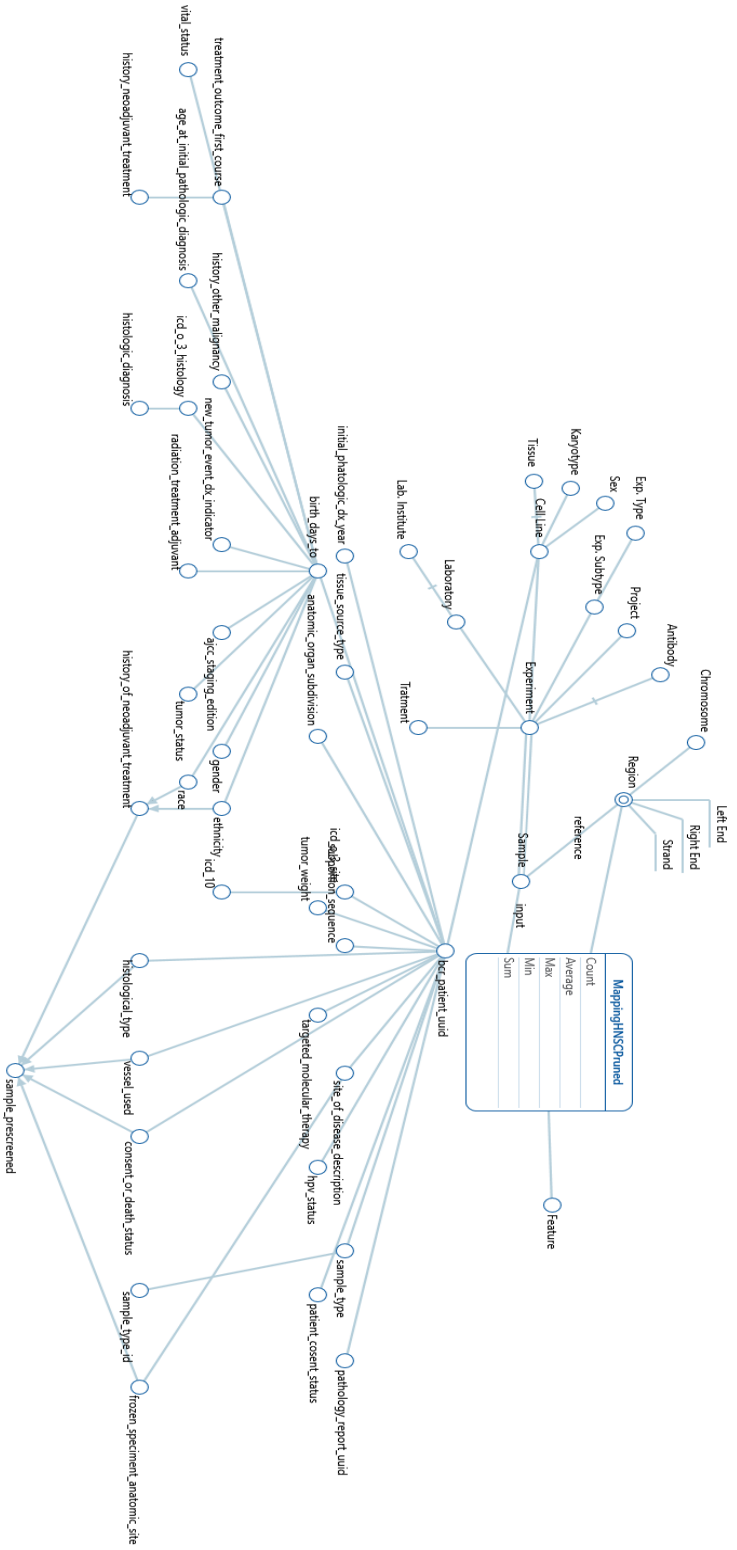


Figura 24 Mapping Cube con gerarchia HNSC pruned

5 Conclusioni

Le tecnologie OLAP e di Data Warehousing sono abbastanza mature e largamente utilizzate in contesti di business, tuttavia non esistono molti progetti sulle loro applicazioni in ambito genomico mentre invece il problema fondamentale che si trova ad affrontare oggi la ricerca biologica è quello di disporre di una quantità immensa e crescente di dati che viene prodotta empiricamente da studi chimici, biochimici e genetico-molecolari e che però non rappresenta una effettiva conoscenza sul modo di funzionare dei sistemi biologici. L'utilizzo di analisi OLAP, modellazione di data warehouse e tecniche di data mining potrebbero invece rilevarsi molto utili in campo genomico trasformando in conoscenza l'enorme quantità di dati disponibile attualmente.

La fase di estrazione dati è stata caratterizzata da un'attenta analisi dei dati, in cui ci si è concentrati sul ridurre la grande mole di dati presenti all'interno del database tcga. In questa fase si è scelto di concentrarsi prevalentemente sui dati di tipo clinico, eliminando anche tutti i dati grezzi e non aggregati presenti all'interno del portale. Il fatto di poter disporre di dati già precedentemente aggregati ha reso la fase di caricamento e denormalizzazione molto più semplice e molto più performante.

La realizzazione e l'implementazione dell'algoritmo per le dipendenze funzionali è stata realizzata utilizzando funzioni di Apache Hive e Apache Spark, ed è stato eseguito sul cluster messo a disposizione dalla facoltà. Le varie esecuzioni dell'algoritmo sono state effettuate seguendo un approccio a "spirale" dove ogni esecuzione veniva sempre più limitata e circoscritta ad una determinata sezione di dati. In questo modo si è potuti giungere a dei risultati chiari e tangibili, validati da una grande quantità di dati, che rappresentano le gerarchie degli attributi del database. L'ultima fase della tesi è stata caratterizzata dall'integrazione dei risultati ottenuti dall'analisi del database con il "mapping cube" basato sulla notazione DFM. Attraverso questo processo è stato possibile collegare la gerarchia del paziente ottenuta con l'analisi ai dati riguardanti le regioni dei campioni di riferimento.

L'inserimento della gerarchia del paziente nel modello multidimensionale permetterà di eseguire analisi di sintesi sugli specifici

attributi dei pazienti. Nel futuro bisognerà quindi cercare di ampliare e completare il modello multidimensionale mapping cube. L'esecuzione dell'algoritmo durante il lavoro di tesi ha fatto emergere non solo la gerarchia del paziente ma anche altre gerarchie, ad esempio la gerarchia relativa ai medicinali e alla loro somministrazione. L'inserimento di questa gerarchia all'interno del mapping cube amplierebbe il raggio delle possibili analisi. Sarà inoltre importante in futuro prevedere analisi anche su attributi di tipo non testuale, come numeri interi, numeri naturali e date. Per fare questo bisognerà modificare tutte le fasi che hanno caratterizzato il lavoro di tesi, dalla fase di caricamento dei dati fino alla fase di scoperta di dipendenze funzionali, il tutto seguendo indicazioni di esperti del settore che dovranno dare una specifica semantica al tipo di dato che si sta analizzando.

6 Bibliografia

[1] «Continuity Raises $10 Milion Series A Round to Ignite Big Data Application Development Within the Hadoop Ecosystem,» 2012.

[2] J. Roman, «The Hadoop Ecosystem Table,» 2014.

[3] M. Rohit, «introducing-mapreduce-part-i,» 4 Gennaio 2013. [Online].

[4] «hadoop.apache.org/,» [Online].

[5] «hortonworks.com,» [Online].

[6] «www.cloudera.com,» [Online].

[7] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker e I. S. Stoica, «SQL and Rich Analytics at Scale,» 2013.

[8] M. Zaharia, «Spark: In-Memory Cluster Computing for Iterative and Interactive Applications,» 2011.

[9] H. Karau, K. Andy, W. Patrick e z. matei, Learning Spark, O'Reilly, 2015.

[10] «spark.apache.org/docs/latest/cluster-overview.html,» [Online].

[11] «spark.apache.org/docs/latest/job-scheduling.html,» [Online].

[12] R. Xin, M. Armbrust e D. Liu, «Introducing DataFrames in Spark for Large Scale Data Science,» 2015. [Online]. Available: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html.

[13] «it.wikipedia.org/wiki/Genomica,» [Online].

[14] A.Fiori, *Classificazione di dati genetici.*

[15] N. Alkharouf, C. Jamison e B. Matthews, «Online analytical processing (OLAP): a fast and effective data mining tool for gene

expression databases,» *Research International,* 2005.

[16] «Using data mining and OLAP to discover patterns in a database of patients with Y-chromosome deletions».

[17] V. Markowitz e T. Topaloglou, «Applyng data warehouse concepts to gene expression data management,» *Proc. BIBE,* 2001.

[18] L. Wang, A. Zhang e M. Ramanathan, «Biostar models of clinical and genomic data for biomedical data warehouse design,» *International Journal of Bioinformatics Research and Applications,* 2005.

[19] D. Havaleshko, H. Cho, M. Conaway, C. Owens, G. Hampton, J. Lee e D. Theodorescu, «Prediction of drug combination chemosensitivity in human bladder cancer,» *Molecular Cancer Therapeutics,* 2007.

[20] X. J. Ma, R. Patel, X. Wang, R. Salunga, J. Murage, R. Desai, T. Tuggle, W. Wang, S. Chu, K. Stecker, R. Raja, H. Robin, M. Moore, D. Baunoch, D. Sgroi e M. Erlander, «Molecular classification of human cancer using a 92-gene real-time quantitative».

[21] J. K. Lee, P. D. Williams e S. Cheon, «Data mining in genomics,» *Clinics in Laboratory Medicine,* 2008.

[22] E. C. Hayden, «Technology: The $1,000 genome,» *Nature,* 2014.

[23] C. Sheridan, «Illumina claims $1,000 genome win,» *Nat. Biotechnol,* 2014.

[24] K. Shvachko, H. Kuang, S. Radia e R. Chansler, «The Hadoop distributed file system,» 2010.

[25] J. Dean e S. Ghemawat, «MapReduce: A flexible data processing tool,» 2010.

[26] A. McKenna, «The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data,»

Genome Res, 2010.

[27] A. Nordberg, «BioPig: a Hadoop-based analytic toolkit for large-scale sequence data,» *Bioinformatics.*

[28] C. Olston, «Pig Latin: A not-so-foreign language for data processing,» 2008.

[29] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jadili, F. Palluzzi, H. Muller e S. Ceri, «GenoMetric Query Language: a novel approach to large-scale genomic data management,» 2015.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar e A. Tomkins, «Pig Latin: A not-so-foreign language for data processing,» 2008.

[31] J. Ekanayake, S. Pallickara e G. Fox, «MapReduce for data intensive scientific analyses,» 2008.

[32] Q. Zou, X. Li, W. Jiang, Z. Lin, G. Li e K. Chen, «Survey of MapReduce frame operation in bioinformatics,» 2013.

[33] R. Taylor, «An overview of the Hadoop MapReduce HBase framework and its current applications in bioinformatics,» 2011.

[34] «GenData2020,» [Online]. Available: http://gendata.weebly.com/.

[35] G. Luciano, *Relaxed FD Discoverer*, 2015.

[36] P. Bohannon, W. Fan, F. Geerts, X. Jia e A. Kementsietsidis, «Conditional Functional Dependencies for Data Cleaning,» 2008.

[37] D. Linstedt, «DV modeling specification,» 2013. [Online]. Available: http://danlinstedt.com/datavaultcat/standards/dvmodeling-specification-v1-0-8/.

[38] V. Jovanovic e I. Bojicic, «Conceptual data vault model,» 2012.

[39] D. Krneta, V. Jovanovic e Z. Marjanovic, «A direct approach to physical data vault design,» 2014.

[40] R. Winter e B. Strauch, «A method for demand-driven information requirements analysis in data warehousing projects,» *proc. HICSS*, 2003.

[41] M. Golfarelli, D. Maio e S. Rizzi, «Conceptual design of data warehouse from E/R schemes,» 1998.

[42] L. Cabibbo e R. Torlone, «A logical approach to multidimensional databases,» 1998.

[43] M. Bohnlein e A. U. Ende, «Deriving initial data warehouse structures from the conceptual data models of the underlying operationa information systems,» 1999.

[44] B. Husemann, J. Lechtenborger e G. Vossen, «Conceptual data warehouse design,» 2000.

[45] C. Phipps e K. C. Davis, «Automating data warehouse conceptual schema design and evaluation,» 2002.

[46] M. R. Jenses, T. Holmgren e T. B. Pedersen, «Discovering multidimnsional structure in relational data,» 2004.

[47] J. Kim, D. Kim, S. Lee, Y. Moon, I. Song, R. Khare e Y. An, «SAMSTRARplus: An automatic tool for generating multi-dimensional schemas from an entity-relationship diagram,» 2009.

[48] B. Vrdoljak, M. Banek e S. Rizzi, «Designing web warehouses from XML schemas,» 2003.

[49] M. Golfarelli, S. Rizzi e B. Vrdoljak, «Data warehouse design from XML sources,» 2001.

[50] M. R. Jensen, T. H. Moller e T. B. Pedersen, «Specifying OLAP cubes on XML data,» 2001.

[51] T. Niemi, M. Niinimaki, J. Nimmenmaa e P. Thanisch, «Constructing an OLAP cube from distributed XML data,» 2002.

[52] S. Anderlik, B. Neumayr e M. Schrefl, «Using domain ontologies as semantic dimension in data warehouses,» 2012.

[53] D. Colazzo, F. Goasdoué, I. Manolescu e A. Roatis, «RDF analytics: lenses over semantic graphs,» 2014.

[54] J. Comput, *Tane: An efficient algorithm for discovering functional and approximate dependencies,* 1999.

[55] M. Senardi, *Data Profiling with Metanome,* www.slideshare.net, 2015.

[56] N. Novelli e R. Cicchetti, «FUN: An efficient algorithm for mining functional and embedded dependencies,» in *Proceeding of the International Conference on Database Theory (ICDT)*, 2001, pp. 189-203.

[57] N. Novelli e R. Cicchetti, «Functional and embedded dependency inference: a data mining point of view,» in *Information Systems*, 2001, pp. 477-506.

[58] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schonberg, J. Zwiener e F. Naumann, «Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms».

[59] H. Yao, J. Hamilton e C. J. Butz, «FD_mine: Discovering functional dependencies in a database using equivalence,» 2002, pp. 729-732.

[60] M. Golfarelli e S. Rizzi, «Data Warehouse design: Modern principles and methodologies,» McGraw-Hill, 2009.

[61] L. Baldacci, M. Golfarelli, S. Graziani e S. Rizzi, «GOLAM: a framework for analyzing genomic data,» 2014.