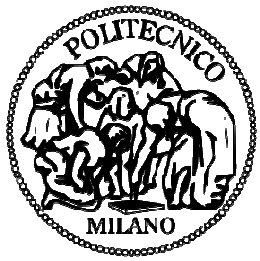
POLITECNICO DI MILANO

V Facoltà di Ingegneria
Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione



Ingegnerizzazione e sviluppo di un'applicazione web dinamica per l'interrogazione efficiente di un grande data warehouse bioinformatico

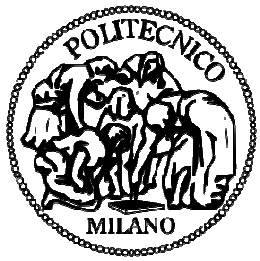
Relatore: Prof. Marco Masseroli, Ph.D

Tesi di Laurea di:
Francesco Pessina
Matr. n. 749982

Anno Accademico 2011-2012

POLITECNICO DI MILANO

V Facoltà di Ingegneria
Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione



Ingegnerizzazione e sviluppo di un'applicazione web dinamica per l'interrogazione efficiente di un grande data warehouse bioinformatico

Laureando: _____
(Francesco Pessina)

Relatore: _____
(Prof. Marco Masseroli)

Anno Accademico 2011-2012

Indice

Indice delle illustrazioni

Indice delle tabelle

Indice dei grafici

1 Sommario

Negli ultimi decenni si è assistito a un rapido sviluppo delle biotecnologie. In particolare lo studio del genoma degli esseri viventi, ha prodotto un'enorme quantità di dati biomolecolari, rendendo necessario lo sviluppo di tecnologie informatiche per la loro gestione ed elaborazione.

Nasce così una nuova disciplina: la bioinformatica. La bioinformatica si dedica all'utilizzo delle tecnologie computazionali per gestire, analizzare e descrivere, in maniera integrata, sistemi biologici complessi al fine di formulare ipotesi sui processi molecolari della vita. La bioinformatica oggi è in rapida evoluzione, grazie soprattutto all'aumento delle capacità di calcolo dei computer e alla possibilità, offerta da Internet, di poter scambiare grandi quantità di dati.

Negli ultimi anni il Dipartimento di Elettronica e Informazione del Politecnico di Milano ha sviluppato una nuova banca dati bioinformatica integrazionale, il Genomic and Proteomic Data Warehouse (GPDW), che persegue l'obiettivo di integrare dati genomici e proteomici provenienti da varie delle maggiori banche dati mondiali bioinformatiche.

Lo scopo di questa Tesi è la progettazione di un'applicazione web che offra uno strumento per l'interrogazione e la visualizzazione dei dati nel GPDW, analogamente a quanto avviene per gli altri database bioinformatici presenti in rete.

Nella realizzazione di questa applicazione si è tenuto conto sopratutto del fatto che il GPDW è un data warehouse molto grande, contenente attualmente quasi due miliardi di tuple; si è rivelato quindi necessario progettare e implementare un algoritmo che generasse query ottimizzate e performanti, per fornire all'utente una risposta in tempi adeguati.

Inoltre il GPDW è frequentemente aggiornato, pertanto l'applicazione deve funzionare anche in seguito a cambiamenti non solo nei dati, ma anche nella struttura del data warehouse, descritta dai suoi metadati. E' stato necessario, quindi, sviluppare degli algoritmi che potessero adattarsi, in modo totalmente trasparente agli utenti, a ogni differente istanza del GPDW: in questo modo il data warehouse può cambiare, ma l'applicazione web rimane la stessa. Ciò ha richiesto un lavoro di sviluppo più attento e accurato, implementando nuove tecniche di controllo utilizzando i dati salvati nei metadati del GPDW, che descrivono la struttura dell'istanza che si sta interrogando.

Si è evidenziata, inoltre, la necessità, per gli utenti, di poter salvare e ricaricare in seguito le proprie interrogazioni effettuate, per poterle modificare e/o eseguire nuovamente.
E' stata quindi progettata un'estensione al database che permettesse la memorizzazione delle query composte dagli utenti, e si sono implementati degli strumenti atti al salvataggio e al caricamento delle stesse. E' stata anche aggiunta la possibilità di proporre agli utenti alcune query ritenute "interessanti" predefinite dagli amministratori.

Dopo il presente sommario, nel secondo capitolo della Tesi è illustrata la struttura del data warehouse di riferimento, il Genomic and Proteomic Data Warehouse, e vengono descritte le principali caratteristiche dell'applicazione web da cui si è partiti per realizzare questo lavoro di Tesi.
Nel terzo capitolo sono illustrati gli obiettivi che ci si è proposti di raggiungere con lo sviluppo della presente Tesi.
Nel quarto capitolo sono descritti gli strumenti e le metodologie tecnologiche utilizzate per il raggiungimento degli scopi enunciati, indicando i software che sono stati utilizzati per la realizzazione della Tesi.
Nel quinto capitolo vengono illustrate tutte le tecniche di reingegnerizzazione dell'applicazione web preesistente. In particolare, vengono descritte le procedure effettuate per separare il livello dei "dati" dal livello "applicativo" dell'applicazione web, descrivendo le strutture dati implementate. Sono poi illustrati i nuovi file di configurazione introdotti e le nuove modalità di connessione al database.
Questa reingegnerizzazione dell'applicazione ha permesso innanzitutto di generare del codice più chiaro e più facilmente modificabile, testabile ed estendibile, fornendo così la possibilità di svolgere in modo più corretto e coerente con le più moderne tecniche di programmazione questo lavoro di Tesi, facilitando inoltre il lavoro di chi dovrà continuare questo sviluppo. Ha poi permesso di aggiungere ulteriori possibilità di configurazione dell'applicazione da parte degli utenti amministratori, senza dover intervenire direttamente nel codice.
Nel sesto capitolo sono illustrate le diverse fasi dello sviluppo della nuova applicazione web. Viene descritto l'algoritmo di generazione dinamica delle query in tutte le sue fasi. Di seguito, vengono esposte le procedure e i criteri di generazione dell'interfaccia di composizione di una query, sia in modalità visuale che in modalità grafica. Segue la descrizione della fase di esecuzione della query, con il processamento dei risultati per fornirne una visualizzazione arricchita all'utente.

Il capitolo si conclude con la descrizione della progettazione e implementazione del sistema di salvataggio e caricamento delle query da parte degli utenti.

Nel settimo capitolo sono esposte le performance dell'applicazione, in termini di tempi di esecuzione delle query generate e di caricamento delle pagine dell'applicazione web, discutendo la validità del trade-off temporale generato dal caricamento dell'interfaccia di composizione della query rispetto a un'interrogazione tradizionale in modo testuale.

Il capitolo si conclude con la relazione del test di usabilità dell'interfaccia dell'applicazione, effettuato osservando nove utenti eseguire alcuni scenari predefiniti per individuare eventuali criticità nel design dell'interfaccia.

La sintesi degli obiettivi raggiunti è inserita nell'ottavo capitolo, mentre il nono capitolo propone alcuni suggerimenti riguardanti i possibili scenari futuri di utilizzo ed estensioni implementabili.

La Tesi termina con il decimo capitolo contenente la bibliografia referenziata.

L'applicazione web risultato di questo lavoro di Tesi è completamente funzionante e già disponibile al pubblico, presso l'url http://www.bioinformatics.polimi.it/GPKB/.

2 Introduzione

2.1 Genomic and Proteomic Data Warehouse (GPDW)

Il Genomic and Proteomic Data Warehouse (GPDW) contiene una grande base di conoscenza di dati biomolecolari e biomedici. Il GPDW nasce da un precedente database bioinformatico, usato nell'applicazione GFINDer (Genomic Function Integrated Discoverer) [2][3]. Tuttavia tale precedente database aveva delle limitazioni progettuali che ne impedivano una facile estensione con l'inclusione di altre tipologie di dati, in particolare di dati proteomici, limitandone fortemente lo sviluppo; si decise quindi di sostituirlo con un nuovo data warehouse, il GPDW. Una dettagliata descrizione della sua struttura e della sua generazione si può trovare in [1].

Il GPDW è stato sviluppato come struttura dati che consente l'integrazione delle informazioni genomiche e proteomiche disponibili e scaricabili via internet da diverse sorgenti (tra cui *Entrez Gene*, *Homologene*, *MINT*, *IntAct*, *Expasy Enzyme*, *GO*, *GOA*, *BioCyc*, *KEGG*, *Reactome* e *OMIM*), e permette inoltre l'evoluzione nel tempo a fronte sia dell'evoluzione del contenuto e della struttura dei dati precedentemente integrati, sia dell'integrazione di dati provenienti da nuove fonti.

Nel GPDW si possono distinguere due livelli di aggregazione dei dati: il livello importato e il livello integrato. Il primo livello contiene tutti i dati direttamente così come sono stati importati dalle loro sorgenti originali. Nel secondo livello, invece, i dati relativi a una stessa feature vengono integrati ed elaborati in modo da verificare la consistenza, tra dati forniti da una stessa o diverse fonti, individuare dati forniti da fonti diverse come relativi alla stessa istanza di feature e riconciliare id non sincronizzati di una stessa istanza di feature forniti da fonti diverse.

Il GPDW è stato realizzato utilizzando come DBMS PostgreSQL (versione 8.4) [4]. All'interno del GPDW sono presenti varie centinaia di tabelle, molte delle quali contenenti svariati milioni di tuple.

Il GPDW ha due schemi principali: lo schema Metadata e lo schema Public.

Lo schema Public contiene le tabelle con i dati importati e integrati e per questo viene aggiornato di frequente non solo per quanto riguarda i dati al suo interno ma, soprattutto, per

lo schema delle tabelle presenti. Questo schema è diviso in moduli in modo da permetterne l'espansione e la ristrutturazione in seguito all'importazione di nuove sorgenti o di nuovi dati.
Per poter gestire questo schema è dunque indispensabile appoggiarsi allo schema Metadata nel quale troviamo le tabelle di metadati relativi alla base di conoscenza, derivati dalla struttura dell'istanza GPDW che descrivono. Dopo un aggiornamento del GPDW il contenuto dello schema dei metadati cambia, riflettendo i cambiamenti avvenuti nei dati pubblici. Lo schema di queste tabelle, invece, non varia; è possibile quindi sfruttarle per ottenere informazioni relative allo schema attuale della base di conoscenza.
Essendo l'estrazione di migliaia di tuple un processo molto oneroso, soprattutto se fra i campi selezionati ve ne sono molti di tipo testo (meno se i campi sono memorizzati come numeri o stringhe di bit), diversi dati sono stati codificati in modo da velocizzare le interrogazioni. Questi dati contengono etichette relative ad altri campi di una stessa tupla. Il GPDW, quindi, include un ulteriore schema, lo schema Flag, che contiene le giuste corrispondenze tra ogni dato codificato e la sua codifica.

2.1.1 La struttura del GPDW: lo schema dei metadati

In questo schema sono raccolti tutti i metadati che descrivono la struttura dello schema Public della specifica istanza del GPDW, indicando, tra le altre cose, quanti moduli sono presenti, da quali tabelle sono composti e descrivendo le associazioni presenti tra tali moduli.
Si descrivono di seguito le caratteristiche di alcune tabelle principali di questo schema:

- ***metadata.feature***: questa tabella contiene tutti i dati relativi alle feature contenute nello schema Public del GPDW, a livello integrato. Tra i dati presenti in questa tabella ricordiamo l'id con cui ogni feature è identificata nello schema Public (*feature_id*), il suo nome, il nome della sua tabella principale nello schema Public ed eventuali sue sottofeatures.
- ***metadata.feature_additional_table***: in questa tabella, a ogni feature_id viene associato il nome di sue eventuali tabelle addizionali di feature nello schema Public.
- ***metadata.source***: in questa tabella sono indicati tutti i dati così come sono stati prelevati dalle sorgenti a cui il GPDW fa riferimento, sono presenti cioè tutti i dati a livello importato. E' indicato il loro source_id (con cui sono identificati nello schema Public), il loro nome e se sono dati direttamente importati nel GPDW o referenziati in altri dati di feature importati nel GPDW.

- ***metadata.source2feature***: questa tabella mette in relazione le feature con le loro sorgenti, indicando a quali feature si riferiscono i dati nel GPDW di ogni sorgente.
- ***metadata.source2feature_additional_table***: qui sono indicati i nomi nello schema Public di tutte le tabelle addizionali di sorgente, relative alla feature di cui forniscono dati.
- ***metadata.feature_association***: in questa tabella sono indicate tutte le associazioni tra coppie di feature, indicando, per ogni associazione, gli id delle due feature in relazione e il nome della tabella nello schema Public che contiene i dati di tale associazione.
- ***metadata.feature_association_additional_table***: qui sono indicate tutte le tabelle addizionali di associazione.
- ***metadata.source2feature_association***: in questa tabella sono indicate tutte le associazioni tra coppie di feature a livello importato, in modo analogo a descritto per la tabella *feature_association*.
- ***metadata.source2feature_association_additional_table***: qui sono indicate tutte le tabelle addizionali di associazione a livello importato.

In Figura 1 si riporta il diagramma logico della parte dello schema dei metadati rappresentante le tabelle citate.

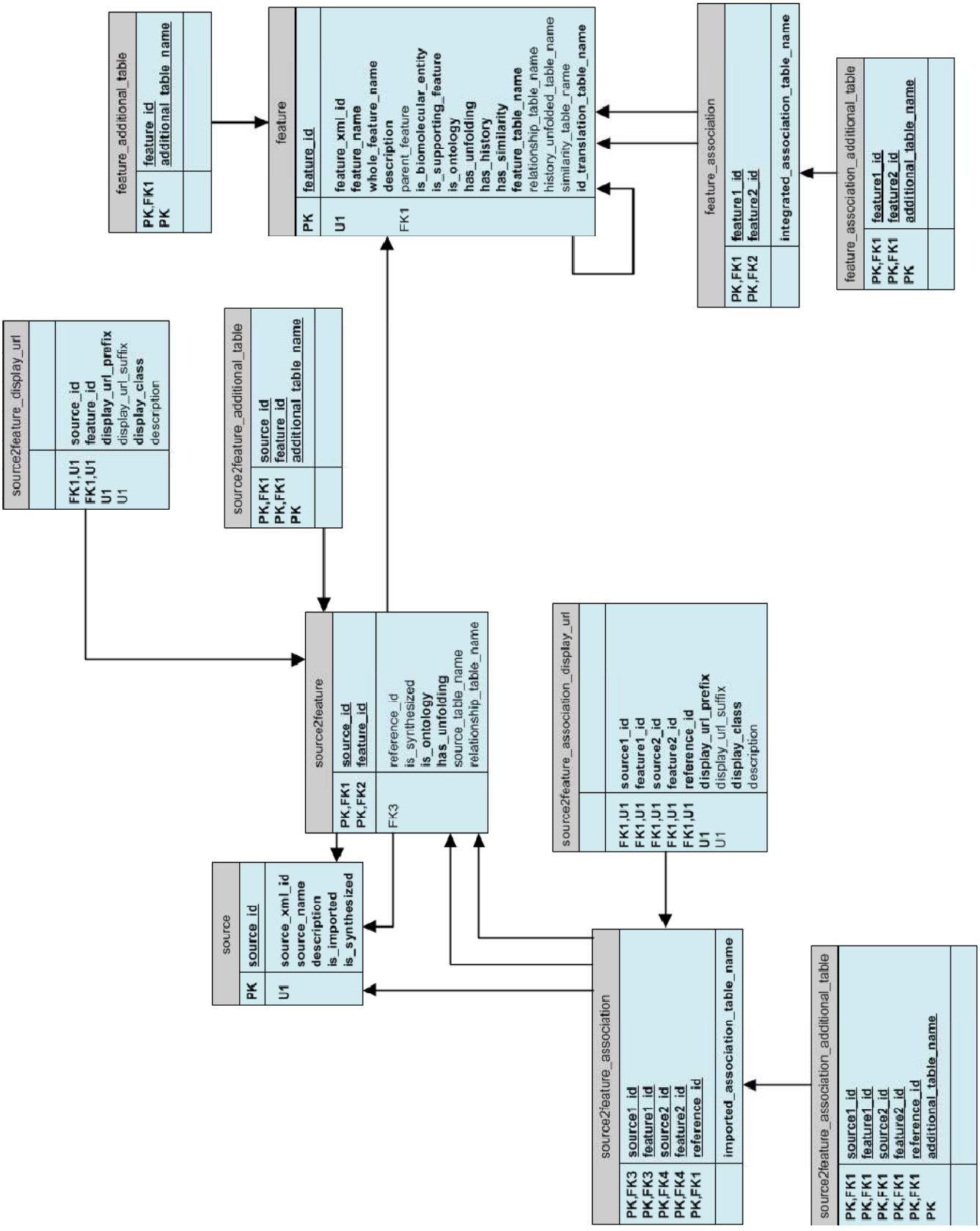


Figura 1 - Diagramma logico di una parte dello schema dei metadati del GPDW

2.1.2 I dati contenuti nel GPDW: lo schema public

Nello schema Public del GPDW troviamo un numero molto elevato di tabelle (411 al momento della stesura di questa Tesi). Queste tabelle sono strutturate in uno schema modulare ben definito, la cui descrizione è salvata nello schema dei metadati (*metadata*) del data warehouse. Attualmente i moduli presenti nel GPDW sono 12.

Ogni modulo contraddistingue una feature (sia essa un'entità biomolecolare o una feature biomedica). Ciascuno di questi moduli possiede diverse tabelle. Tra queste, quelle definite "principali" sono le tabelle di ***feature*** *(tabella rosa più grande* in Figura 2) che contengono tutti i dati relativi alle entità e alle caratteristiche biomolecolari. A oggi, le feature salvate nel GPDW sono:

- **entità biomolecolari**:
  - dna_sequence
  - gene
  - protein
  - transcript
- **feature biomediche**:
  - biological_function_feature
  - clinical_synopsis
  - enzyme
  - gene_expression_feature
  - generic_disorder
  - pathway
  - protein_fam_dom
  - small_molec

I moduli di entità biomolecolari contengono una quantità molto grande di dati: le loro tabelle principali di feature, infatti, contengono milioni di tuple. La più piccola tra loro, *dna_sequence*, a oggi possiede circa 4 milioni di tuple; la più grande, "protein" ne possiede più di 62 milioni. I moduli di feature biomediche, invece, sono più piccoli: la tabella di feature più grande tra loro, *biological_function_feature*, contiene 36.963 tuple.

Ogni modulo, oltre alla tabella principale di feature, possiede alcune tabelle secondarie, che aggiungono informazioni alla feature in oggetto.
Le tabelle secondarie sono: addizionali di feature (*additional_feature*), di sorgente (*source*), addizionali di sorgente (*additional_source*) e di *id_translation*. Nel dettaglio:

- ***Tabelle addizionali di feature (additional_feature)***: le tabelle addizionali di feature contengono dati addizionali, cioè tutti quei dati con cardinalità maggiore di uno, rispetto a quelli di feature. Per trovare queste tabelle bisogna effettuare una ricerca nella tabella *feature_additional_table* presente nello schema dei metadati. (*Tabelle rosa "piccole"* in Figura 2).
- ***Tabelle di id translation***: le tabelle di id translation sono, come indicato dal nome, tabelle di traduzione. Esistono per ogni entità biomolecolare e feature biomedica nel GPDW e forniscono, per ogni id di una feature nel GPDW, la mappatura con i suoi eventuali vari id obsoleti o alias. Per ogni id obsoleto, in questa tabella c'è la mappatura con il proprio corrispondente id attuale. Se l'id ha degli alias, questa tabella fornisce tutti i suoi alias importati nel GPDW. In questo caso per trovare la tabella di id translation (una per ogni feature) bisogna cercare nella colonna *id_translation_table_name* della tabella *metadata.feature* (*tabella azzurra* in Figura 2).

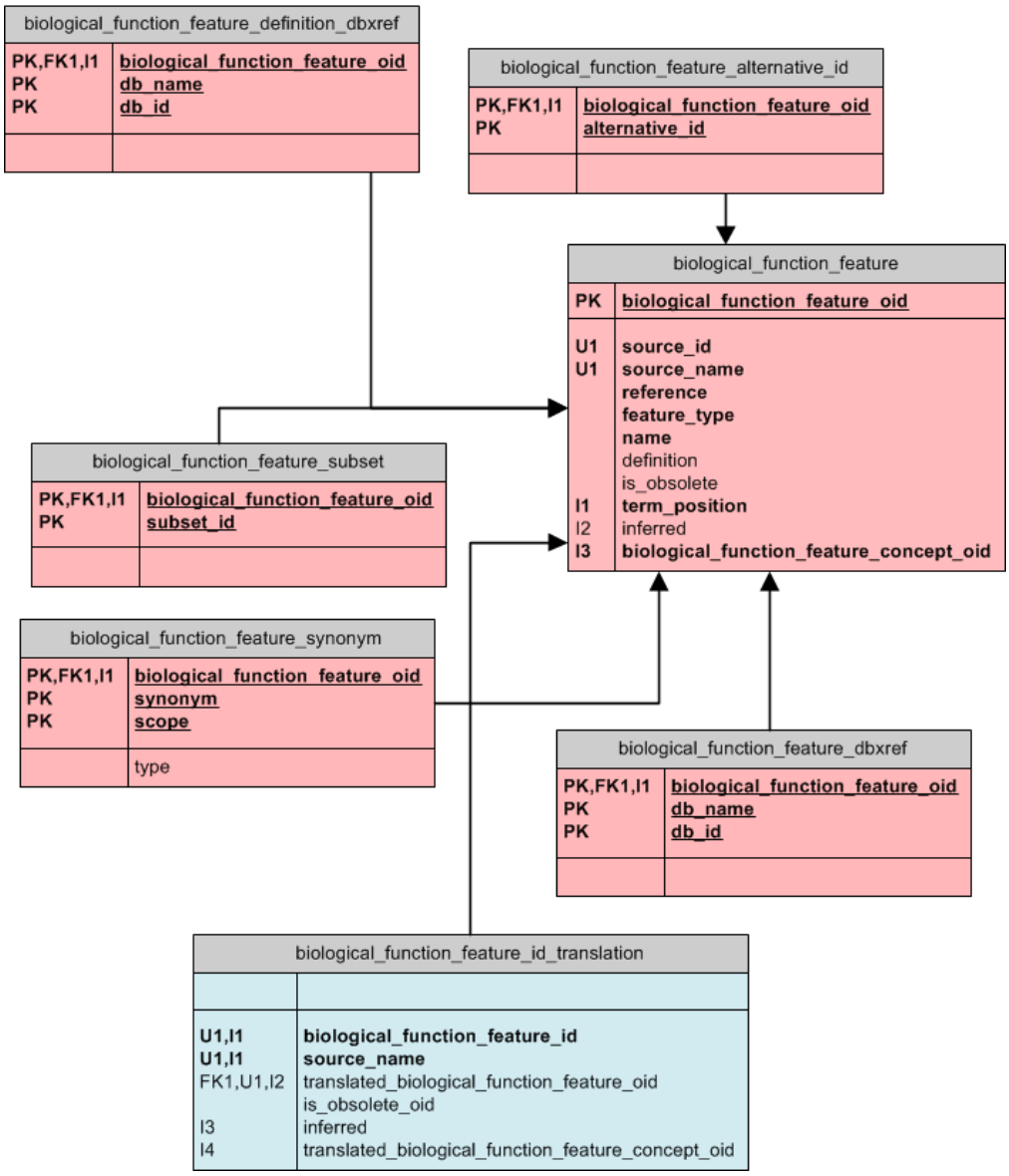


Figura 2 – Diagramma logico di parte dello schema Public relativo alla feature "biological_function_feature": tabella principale di feature, tabella di id_translation e tabelle addizionali di feature

- ***Tabelle di sorgente (source)***: le tabelle di sorgente (source) contengono i dati di sorgente (cioè prelevati direttamente dalle sorgenti, a livello importato) per la feature selezionata. Per trovare queste tabelle bisogna cercare nella tabella *source2feature* presente nello schema dei metadati. (*tabella bianca più grossa* in Figura 3)
- ***Tabelle addizionali di sorgente (additional_source)***: le tabelle di addizionali di sorgente contengono i dati addizionali di sorgente, cioè tutti quei dati con cardinalità maggiore di uno, per ogni sorgente. Come nel caso precedente troviamo i nomi di queste in *source2feature_additional_table*, nei metadati. (*tabelle bianche più piccole* in Figura 3)

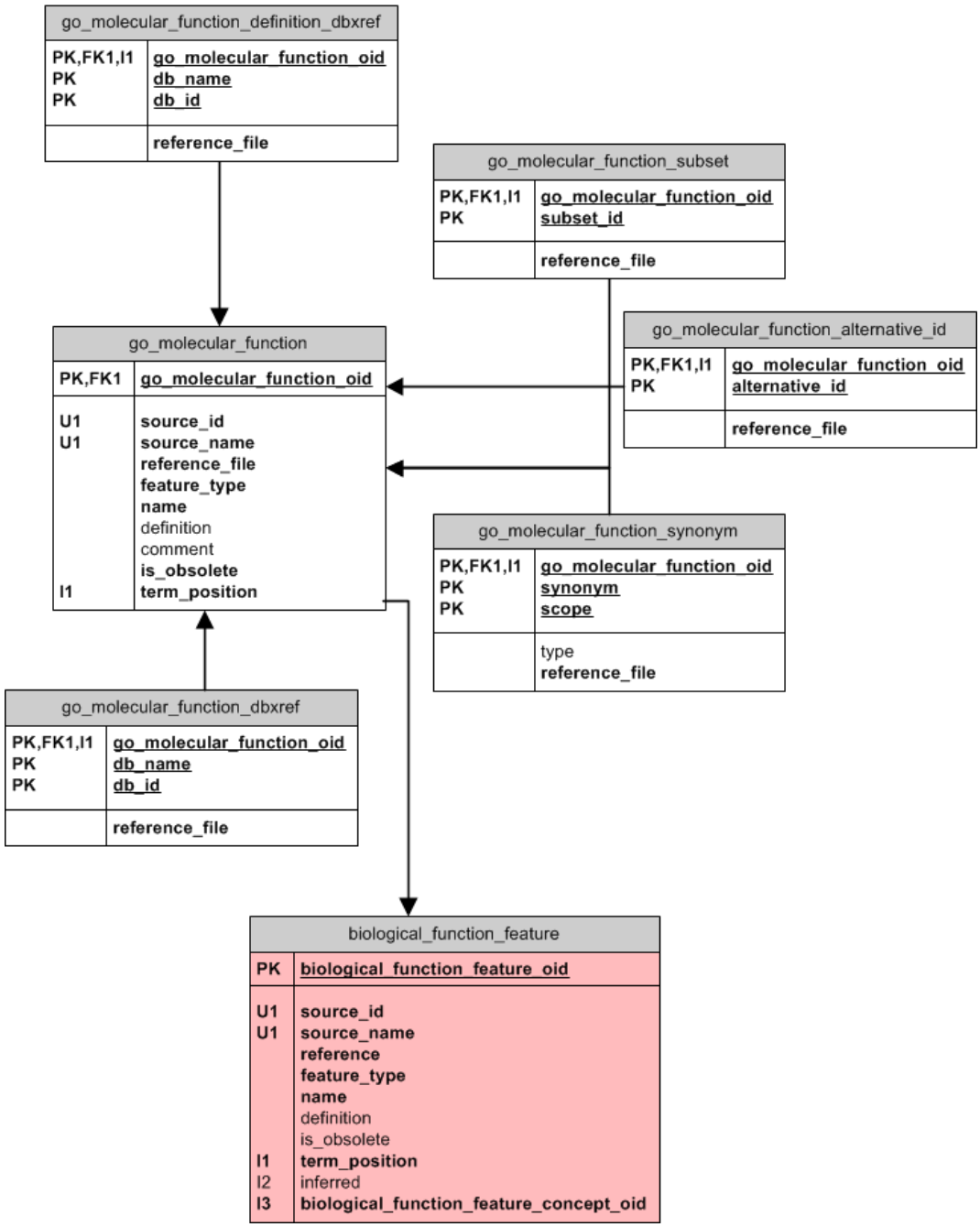


Figura 3 – Diagramma logico di parte dello schema public della feature "biological_function_feature": tabella di sorgente e addizionali di sorgente

Le tabelle di ogni modulo del GPDW sono inserite all'interno di una gerarchia. Questa gerarchia può essere descritta con un **Direct Acyclic Graph (DAG)**, come rappresentato in Figura 4.

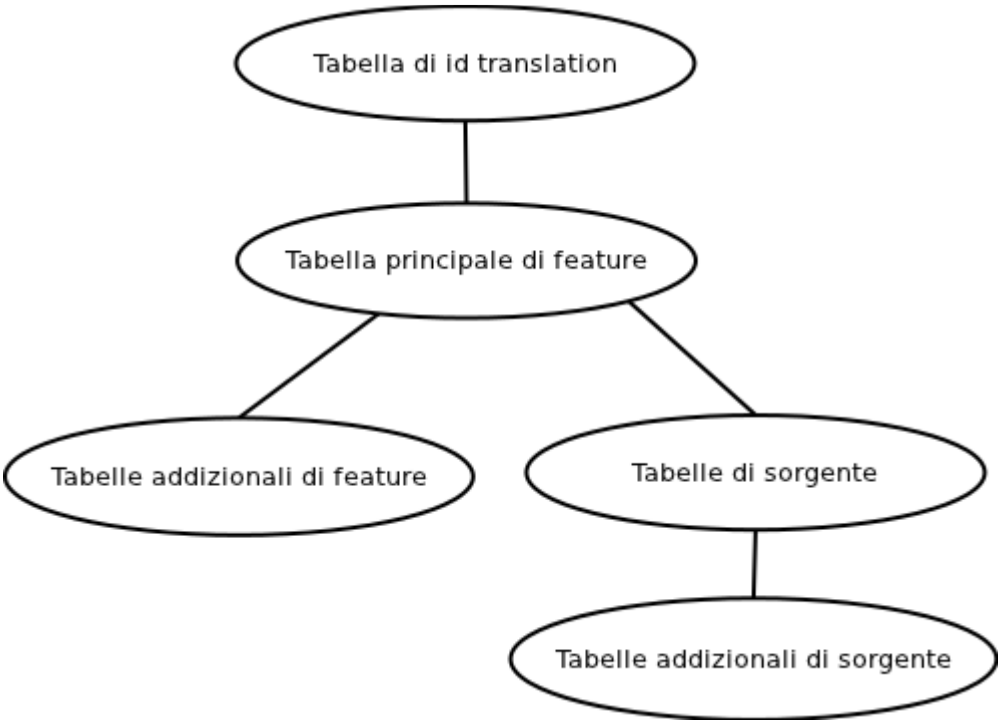


Figura 4 - Direct Acyclic Graph delle tabelle di un modulo del GPDW

Ogni tipo di tabella, quindi, ha un suo "padre":

- Tabella di **feature**: il padre è la tabella di *id_translation*.
- Tabella **addizionale di feature**: il padre è la tabella di *feature*
- Tabella di **sorgente**: il padre è la tabella di *feature*
- Tabella **addizionale di sorgente**: il padre è la tabella di *sorgente*
- Tabella di **id_translation**: il padre è la tabella di *associazione* nel caso di multipla feature, oppure nessuno *(null)*.

Ogni tabella figlio, per costruzione, possiede al massimo tante tuple quante la tabella padre, eccezion fatta per le tabelle addizionali.

Alcuni di questi moduli, inoltre, possono rappresentare delle "sottofeature". Questo significa che nelle tabelle di questi moduli sono salvati dati di più di una o più sottofeature, che sono racchiuse all'interno dello stesso modulo. La distinzione tra queste sottofeature si può trovare nella tabella *metadata.feature*, dove sono indicate tutte le feature nel GPDW (principali e "sottofeature") e per ogni sottofeature è indicata la feature principale a cui appartiene. Nello schema Public, all'interno di ogni modulo che rappresenta più sottofeature, tutti i dati relativi a una data sottofeature si possono ottenere filtrando le tuple della tabella di feature principale

secondo il valore del campo *feature_type*, che deve assumere il valore del nome della sottofeature cercata.

Possono esistere, inoltre, relazioni tra due feature. Queste relazioni tra feature, nel GPDW, sono modellate attraverso delle ***tabelle di associazione*** *(association)* (*tabelle marroncine grandi* in Figura 5). Tutte le associazioni presenti nel GPDW sono rappresentate nei metadati nella tabella *feature_association*. Queste tabelle di associazione descrivono collegamenti tra moduli di feature. Ogni tabella di associazione, definita "principale", possiede delle tabelle secondarie:

- ***Tabelle addizionali di associazione* (*additional_association*)**: le tabelle addizionali di associazione contengono dati addizionali, cioè tutti quei dati con cardinalità maggiore di uno rispetto a quelli di associazione (*tabella marroncina piccola* in Figura 5). Queste tabelle sono rappresentate nei metadati nella tabella *feature_association_additional_table*.
- ***Tabelle di associazione importate (imported_association)***: le tabelle di associazione importate contengono i dati importati di associazione (*tabella gialla grande* in Figura 5). Per trovare le tabelle di associazione importate è necessario effettuare una ricerca nei metadati nella tabella *source2feature_association*, prelevando il valore del campo *imported_association_table_name*.
- ***Tabelle addizionali di associazione importate (imported_additional_association)***: le tabelle addizionali di associazione importate contengono dati addizionali, cioè tutti quei dati con cardinalità maggiore di uno rispetto a quelli di associazione importata (*tabella gialla piccola* in Figura 5). Queste tabelle sono indicate nei metadati, nella tabella *source2feature_association_additional_table*, con il valore del campo *additional_table_name*.

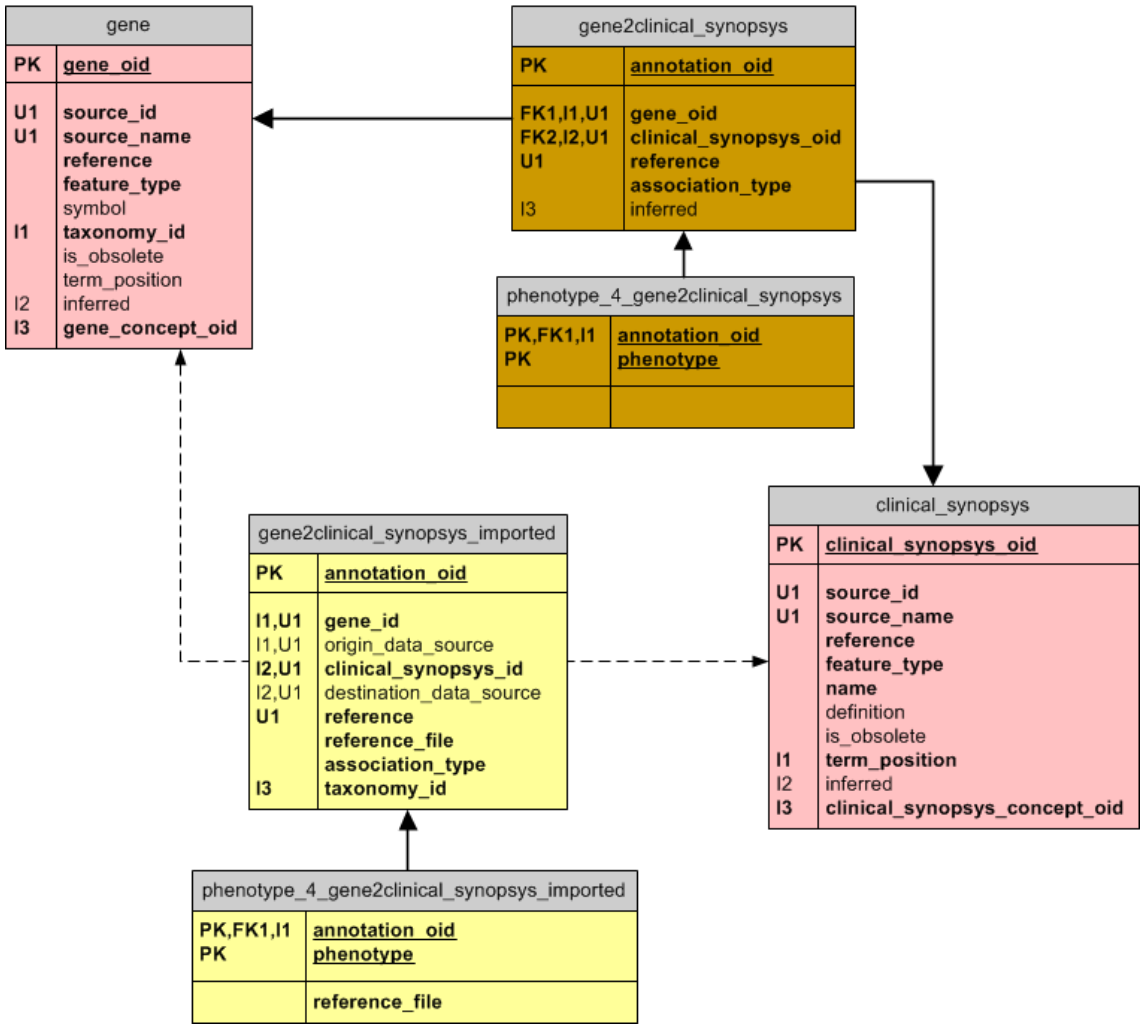


Figura 5 - Diagramma logico di parte dello schema public relativa all'associazione tra la feature "gene" e la feature "clinical_synopsys"

Analogamente a quanto detto per le tabelle appartenenti ai moduli rappresentati una feature, anche le tabelle di associazione possono essere ordinate in una gerarchia, rappresentata dal DAG in Figura 6.

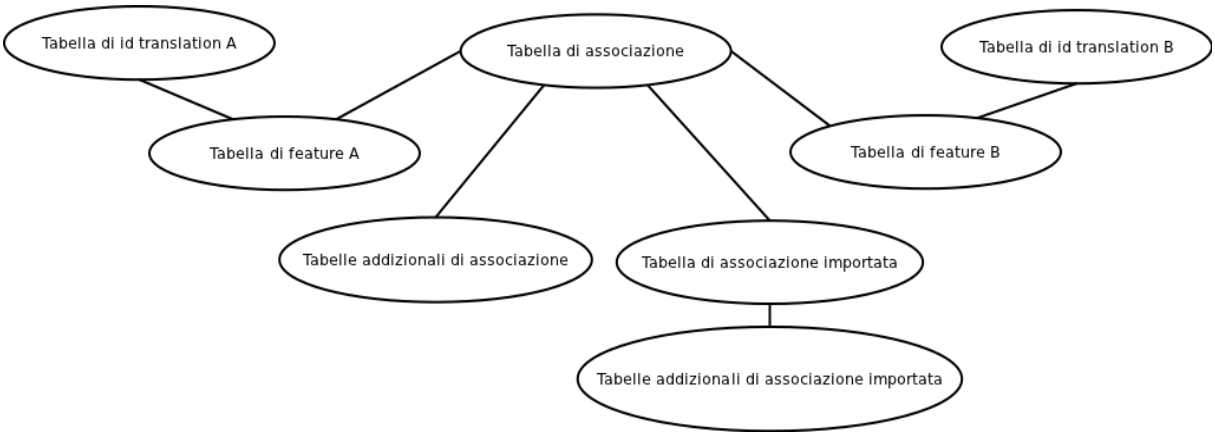


Figura 6 - DAG della gerarchia tra tabelle di associazione tra due moduli di feature

Come mostrato in figura, anche ognuna di queste tabelle può avere una tabella "padre":

- Tabella di **associazione**: queste tabelle non possiedono tabelle padre
- Tabella **addizionale di associazione**: il padre è a tabella di *associazione* corrispondente
- Tabella di **associazione importata**: il padre è la tabella di *associazione* corrispondente
- Tabella **addizionale di associazione importata**: il padre è la tabella di *associazione importata* corrispondente

Inoltre, quando coinvolte in una associazione, le tabelle di feature possiedono due "padri": la loro tabella di id translation corrispondente e la tabella di associazione.
Anche in questo caso, per costruzione, ogni tabella figlio possiede al massimo tante tuple quante la tabella padre, eccezion fatta per tutte le tabelle addizionali e le tabelle di associazione importata, che possiedono un numero maggiore di tuple rispetto alla loro corrispondente tabella di associazione.

2.1.3 Campi codificati e tabelle di Flag

Visto il gran numero di tuple contenute nello schema *Public* del GPDW, si è deciso di codificare alcuni campi tramite tabelle di ***Flag***. In particolare, sono stati codificati i valori dei campi che contengono etichette relative ad altri campi di una stessa tupla.

Nello schema Public, quindi, sono salvati i valori codificati di questi campi. Per questo, se si mostrasse il contenuto di una tabella dello schema Public senza svolgere nessuna operazione, l'utente si troverebbe davanti ad una serie di codici per lui poco significativi.
Per risolvere questo problema è necessario sostituire il valore codificato con la sua controparte non codificata, effettuando una ricerca nella tabella ***flags*** dello schema flag del GPDW.
La tabella *flags* contiene al suo interno, per ogni campo codificato presente nello schema public, l'indicazione di quale tabella dello schema flag andare a interrogare per ottenere la coppia codice/valore reale di tale campo codificato. Questa indicazione è salvata nel campo *specific_flag_table_name* della tabella *flags.* In questo modo, in fase di visualizzazione dei risultati si può sostituire il valore codificato con quello testuale presente nella corrispondente tabella.

2.2 Precedente applicazione web per l'interrogazione del GPDW

Questo lavoro di Tesi non è partito da zero, ma ha sviluppato un'applicazione precedentemente esistente [5].

L'applicazione web da cui si è partiti per realizzare questo lavoro di Tesi è stata sviluppata in linguaggio Java (per quanto riguarda la parte delle classi servlet) e JSP (per le pagine di interfaccia con l'utente).

Le potenzialità di questa applicazione prevedevano la possibilità di effettuare una ricerca all'interno dello schema Public del GPDW limitato a una singola feature, tramite una classica interfaccia di tipo form HTML.

Tutti i menu erano semplici menu standard HTML, che permettevano un'esperienza d'uso piuttosto limitata e poco *user-friendly*.

La visualizzazione dei risultati prevedeva già l'utilizzo del plugin jQuery Datatables, senza però la possibilità di filtrare i risultati stessi.

2.2.1 Configurazione dell'applicazione

La web application dispone di un file xml chiamato web.xml [6], detto anche descriptor deployment, attraverso il quale è possibile definire le informazioni di configurazione. Il root element del file web.xml è <web-app>, all'interno del quale sono stati inseriti i seguenti elementi top-level:

- ***display-name***: definisce il nome della web application, nel nostro caso è "GPKB";
- ***description***: contiene una descrizione relativa al sito;
- ***context-param***: permette di definire dei parametri di inizializzazione per la web application (o meglio del **ServletContext**) ai quali è possibile accedere mediante i metodi *getInitParameter()* e *getInitParameterNames()* della classe `ServletContext`. Per ogni parametro è specificato il nome, il valore ed una descrizione opzionale. Un'esempio di context-param dichiarato nella web application è riportato in Tabella 1.

```
<context-param>
    <description>Database information file</description>
    <param-name>config</param-name>
    <param-value>WEB-INF/config.xml</param-value>
</context-param>
```

Tabella 1 - Configurazione del context-parameter "config"

Il valore del parametro *config* è il percorso relativo del file *config.xml* contenente le credenziali per la connessione al data warehouse GPDW e al database GPDW_web. Nel file sono presenti anche altre informazioni necessarie per l'installazione e il restore del database GPDW_web;

- ***listener***: contrassegna gli event listener. L'applicazione dispone di un Session Listener chiamato *HttpSessionListener* i cui metodi sono: *sessionCreated(HttpSessionEvent)*, eseguito nell'istante in cui viene creata una nuova sessione, e *sessionDestroyed(HttpSessionEvent)* eseguito invece quando una sessione scade. Il listener *HttpSessionListener* a ogni nuova sessione controllerà lo stato del GPDW e del database GPDW_web, mentre quando la sessione scade provvederà a deregistrare i driver JDBC, utilizzati per la connessione ai database, in modo da evitare perdite di memoria.
- ***servlet***: serve per definire le servlet dell'applicazione.
- ***servlet-mapping***: fornisce un URL di default relativo ad ogni servlet dichiarata, permettendo al server di localizzarle.
- ***session-config***: utilizzato per definire i parametri di sessione come la durata in minuti del timeout, nel nostro caso 30 minuti;
- ***welcome-file-list***: contiene una lista ordinata di <welcome-file> che definiscono l'ordine di risposta quando una richiesta HTTP corrisponde ad una directory della web application;
- ***error-page***: permette di definire le pagine da visualizzare nel caso in cui si verifichi un errore. Nella applicazione sono definite due pagine di errore, una per l'errore 404 (pagina non trovata) e l'altra per l'errore 500 (errore interno al server).

Il file web.xml non è l'unico file di configurazione presente nella web application.
Ci sono infatti altri due file XML:

- ***config.xml***: file di configurazione della connessione. Per approfondire vedere il paragrafo 5.3.1;

- ***query.xml***: questo file contiene le query statiche, i template sql e i prepared statement utilizzate dalle varie servlet. L'adozione di questo file xml consente di mantenere separato il codice dalle query eseguite, rendendo semplice e immediato qualunque modifica senza dover intaccare la struttura della web application;

Ogni file dispone di una classe adibita al recupero delle informazioni contenute in esso.
Queste classi, tutte contenute nel package "xml", sono:

- xml.XMLDb che si occupa del file config.xml;
- xml.Query funzionale al file query.xml;

Le classi sopraccitate, prima di procedere alla lettura, verificano che i file siano validati secondo i rispettivi XML Schema, invocando il metodo *isValidated()* della classe *xml.ValidateXML*.
Tutte e tre le classi prevedono un costruttore avente come parametro di ingresso il percorso assoluto del file XML che devono gestire. Nel file web.xml sono stati salvati i percorsi relativi dei tre file come context-parameter. Il metodo *sessionCreated(HttpSessionEvent)* della classe *HttpSessionListener* si occuperà di ricavare il percorso assoluto e di caricarlo in sessione. Quando una servlet avrà bisogno di instanziare una delle tre classi sopraccitate basterà che prelevi il percorso assoluto necessario direttamente dalla sessione.

Per estendere le funzionalità della web application, è stato necessario definire nuovi parametri di configurazione, in modo analogo a quelli precedentemente descritti. Queste nuove possiblità di configurazione verrano esposte nella sezione 5.4.

2.2.2 Gestione utenti: database GPDW_web

Per la gestione degli utenti, l'applicazione si avvale del database "***GPDW_Web***", progettato per memorizzare i dati personali degli utenti e registrare tutte le operazioni effetuate da questi ultimi. Brevemente, si riporta lo schema concettuale del database, riguardante le gestione degli utenti.

Schema concettuale

Nella progettazione concettuale del GPDW_web è stata definita un'entità USER adibita alla registrazione dei dati personali degli utenti. Gli attributi dell'entità **USER**, oltre ai dati anagrafici e username e password, sono:

- ***Type***: l'utente registrato può essere di tre tipi: Standard, Amministratore o Avanzato. L'attributo *Type* vale 0 nel caso in cui l'utente sia un Amministratore, 1 se è Standard e 2 se è Avanzato;
- ***Registration Timestamp***: memorizza la data e l'ora di registrazione dell'utente.
- ***Is Enabled***: è un attributo booleano, vale TRUE se l'utente è abilitato all'accesso del sito o FALSE altrimenti;
- ***Last Connection***: che serve per memorizzare data e ora dell'ultima operazione effettuata dall'utente in questione;
- ***Registration code***: al momento della registrazione viene generato un codice alfanumerico casuale. Serve per verificare l'email dell'utente e confermare quindi la sua registrazione;
- ***Working during switch over***: indica se l'utente era connesso durante l'aggiornamento del GPDW. Anche questo attrubuto servirà per un prossimo sviluppo.

Un utente Amministratore ha la facoltà di promuovere un utente da Standard ad Avanzato. Per questo è prevista l'associazione HAS ENABLED contenente l'attributo ***Upgrading timestamp***, indicante ora e data della abilitazione ad utente avanzato. Un Amministratore può promuovere un numero illimitato di utenti, ma un utente non può essere promosso da più di un Amministratore.

Gestione degli utenti

Come detto nella sezione precedente, nell'applicazione gli utenti possono essere di tre tipi: *Amministratore*, *Standard* e *Avanzato*.

Le seguenti funzionalità sono utilizzabili esclusivamente da un utente Amministratore:

- Abilitazione degli utenti all'accedere al sito;
- Revocare il diritto d'accesso ad un utente;
- Visualizzare il contenuto del database GPDW_web.

Il motivo per cui esiste un solo Amministratore è dato dal fatto che il primo utente che si registra al sito diventa Amministratore, infatti colui che procede all'installazione del

GPDW_web e dell'application web necessariamente sarà anche l'amministratore del server; quindi subito dopo aver effettuato il deploy dell'applicazione, l'amministratore si registrerà al sito ottenendo automaticamente i privilegi previsti per l'utente Amministratore.

Gli utenti che si registreranno successivamente saranno di tipo Standard ed automaticamente abilitati all'accesso, salvo disabilitazione successiva da parte dell'Amministratore.

Altre peculiarità dell'utente Amministratore sono:

- ***upgrading_admin*** ha lo stesso valore del campo *username*, questo perché l'Amministratore ha promosso sé stesso;
- ***upgrading_timestamp*** coincide con il campo ***registration_timestamp***, in quanto contestualmente alla registrazione avviene anche la concessione dei privilegi di Amministratore.

I campi *upgrading_admin* e *upgrading_timestamp* sono utilizzati per denotare gli utenti Standard, ma si può ragionevolmente pensare che un utente Amministratore è anche un utente Standard.

2.2.3 Interfaccia di composizione delle query

In Figura 7 viene riportato uno screenshot dell'interfaccia di composizione della query dell'applicazione precedente.

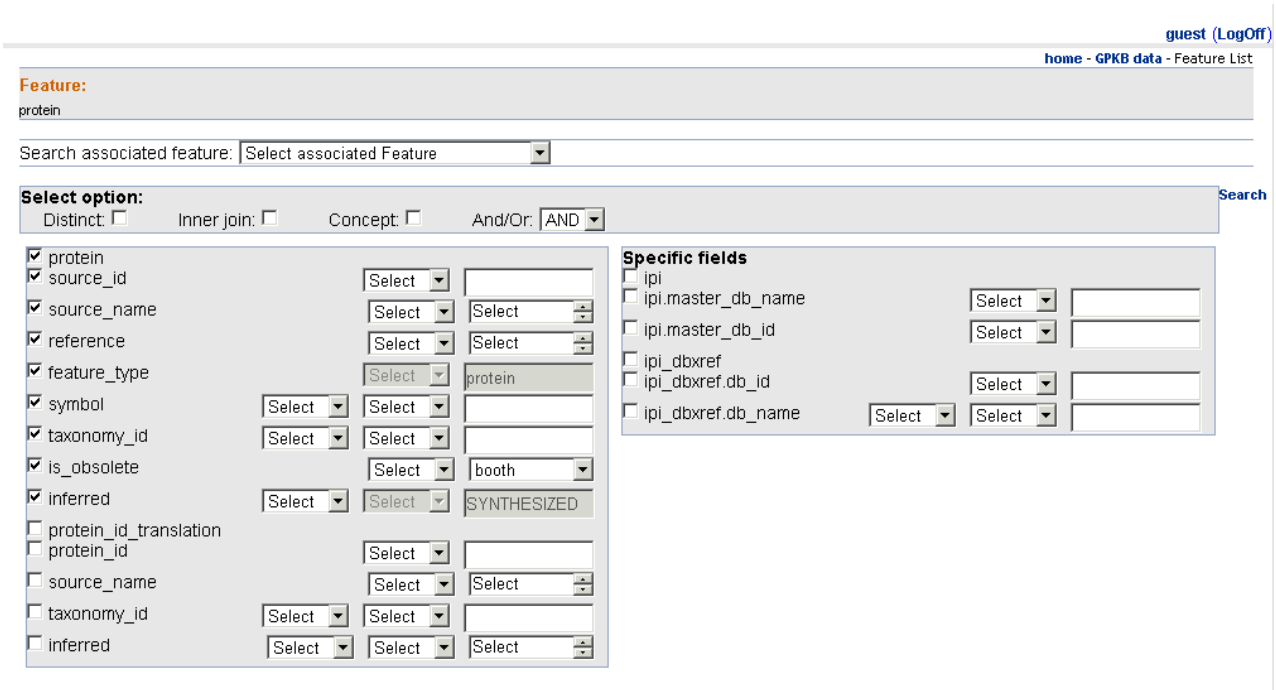


Figura 7 - Interfaccia di composizione query della precedente applicazione

I menù sono tutti normali select o input html. In particolare, i select di tipo multiple (cioè che permettono la selezione di più di un valore alla volta), permettono di visualizzare un solo valore alla volta, rendendo così la loro selezione molto difficoltosa.
Inoltre, gli attributi delle feature non sono divisi secondo le loro tabelle.

2.2.4 Visualizzazione dei risultati

In Figura 8 viene riportato uno screenshot della pagina di visualizzazione dei risultati della precedente applicazione.

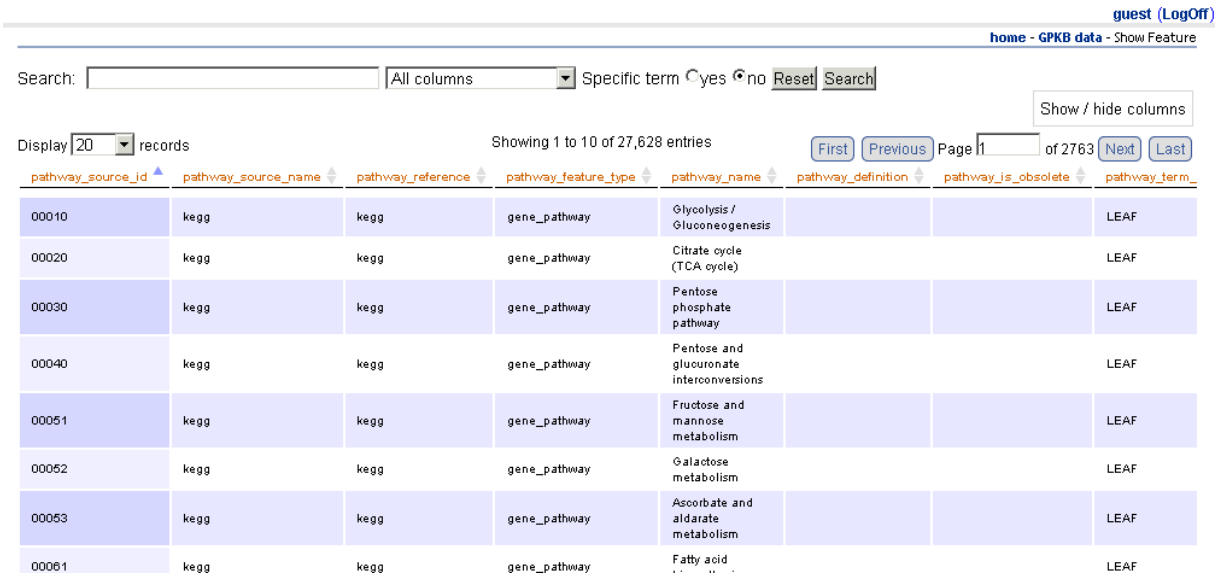


Figura 8 - Pagina di visualizzazione dei risultati della precedente applicazione

La visualizzazione è tabellare, paginata. E' possibile ordinare i dati secondo il contenuto di ciascuna colonna, in maniera ascendente o discendente. E' inoltre possibile nascondere dalla visualizzazione alcune colonne, tramite l'opzione "Show/hide columns".
La ricerca all'interno dei dati visualizzati permette di ricercare valori testuali all'interno di una o tutte le colonne visualizzate.

3 Obiettivi della Tesi

Il Web è uno strumento indispensabile per la diffusione e la condivisione della conoscenza, e, ogni giorno, si moltiplicano le applicazioni e i servizi che offrono nuove e interessanti possibilità per gli utenti. Le applicazioni e i servizi web, grazie al rapido sviluppo della tecnologia, utilizzano e generano un numero sempre maggiore di dati. Grazie alle possibilità e potenzialità di Internet la Bioinformatica ha potuto svilupparsi notevolmente negli ultimi anni. Questa disciplina nasce per fornire validi strumenti per l'elaborazione dell'enorme mole di dati biomolecolari che la ricerca biotecnologica produce e sono resi disponibili in numerose sorgenti dati distribuite. I risultati dell'elaborazione bioinformatica di tali diversi dati concorrono alla creazione della conoscenza che può essere utilizzata per formulare e validare ipotesi, ed eventualmente scoprire nuova conoscenza biomedica. Integrare le informazioni fornite da molteplici fonti è quindi fondamentale per scienziati e ricercatori. Per questo i più importanti enti e istituti internazionali bioinformatici permettono di consultare le proprie banche dati attraverso il Web.

Lo stesso vale anche per il Dipartimento di Elettronica e Informazione del Politecnico di Milano che ha progettato e realizzato il Genomic and Proteomic Data Warehouse. Il GPDW, essendo un data warehouse che integra dati provenienti da diverse banche dati bioinformatiche, costituisce una fonte importante nello scenario bioinformatico.

E' stato quindi dotato in passato di un'iniziale applicazione web che ne permettesse la facile ed efficiente consultazione via rete.

L'obiettivo principale di questa Tesi è migliorare e sviluppare ulteriormente questa applicazione web, estendendone le funzionalità e la fruibilità e rendendola flessibile in modo tale da permetterne l'adattamento automatico a diverse istanze del data warehouse: il contenuto del GPDW, infatti, varia nel tempo, ma grazie alla sua struttura modulare descritta nei propri metadati, è possibile generare un'architettura software che possa adattarsi a ogni diversa versione del data warehouse e ne permetta un'interrogazione efficiente.

Per raggiungere questo scopo principale, gli obiettivi prefissati della Tesi sono:

- ***Separazione livello "dati" e livello "applicativo"***: per permettere uno sviluppo dell'applicazione snello e coerente con le più aggiornate modalità di progettazione, è necessario separare il livello "dati" dell'applicazione (cioè il modo di prelevare i dati dal

database) dal livello applicativo (cioè il modo di rappresentare ed elaborare i dati prelevati dal database all'interno dell'applicazione).

- ***Algoritmo di generazione dinamica delle query***: l'applicazione web precedentemente sviluppata genera query per singola feature in modo statico e poco flessibile. Obiettivo di questa Tesi è progettare e implementare un algoritmo di generazione dinamica delle query, che permetta di generare delle interrogazioni ottimizzate secondo ciò che l'utente ha richiesto, indipendentemente dall'istanza del GPDW che si va a interrogare. Sfruttando infatti la struttura del GPDW, è possibile generare delle query sia sintatticamente sia semanticamente corrette anche senza includere tutte le tabelle che sarebbe necessario includere seguendo il percorso di join indicato dai vincoli di chiave esterna del database.
- ***Query multifeature***: l'applicazione web precedentemente sviluppata fornisce la possibilità di effettuare ricerche soltanto su di una singola feature. Obiettivo di questa Tesi è estendere la possibilità di effettuare ricerche all'interno del GPDW anche su due o più features, sia ideando nuovi algoritmi di generazione dinamica delle query sia opportunamente modificando l'interfaccia grafica dell'applicazione web.
- ***Modalità di conteggio dei risultati***: l'applicazione web precedentemente sviluppata utilizza una modalità di conteggio "approssimato" dei risultati (tuple) trovati da una query, utilizzando la funzione SQL *explain* di PostgreSQL. Si vuole introdurre una modalità di conteggio che permetta di ottenere il numero esatto di tuple ritornate in un tempo ragionevole, o comunque che non allunghi i tempi dell'esperienza d'uso dell'utente.
- ***Inserire possibilità di salvataggio e caricamento delle query***: si vuole fornire la possibilità agli utenti di poter salvare e ricaricare in seguito le query da loro composte, per poterle modificare oppure rieseguire successivamente. Inoltre, si vuole poter fornire a tutti gli utenti una serie di query predefinite che tutti possano eseguire. Per questo, è necessario modificare il database *GPDW_web*, già presente nell'applicazione e costruito per raccogliere i dati relativi agli utenti registrati, inserendo nuove tabelle atte a questo scopo.
- ***Miglioramento interfaccia già esistente***: l'interfaccia per la composizione della query presente nell'applicazione web preesistente si è rivelata poco usabile. Di conseguenza, è necessario modificare questa interfaccia, consentendo nuove e maggiori possibilità di ricerca e di scelta e definizione di filtri di ricerca, sia in fase di composizione della query che in fase di visualizzazione dei risultati.

- ***Realizzazione interfaccia per query multifeature***: per permettere una buona esperienza d'uso nella composizione di query su di più di due features, è necessario implementare un nuovo tipo di composizione della query, definito "grafico", in quanto permetta la selezione delle features su cui effettuare la ricerca tramite un grafo generato in base ai metadati del GPDW.

4 Strumenti e metodologie tecnologiche

4.1 DBMS PostgreSQL

Il DBMS (DataBase Management System) utilizzato per la gestione dei dati è PostgreSQL. PostgreSQL [4] è un potente DBMS relazionale ad oggetti di tipo open source ed è sviluppato da una nutrita comunità. Più di 15 anni di sviluppo attivo e di architetture testati hanno permesso a PostgreSQL di conseguire un'ottima reputazione per l'affidabilità, l'integrità dei dati e la precisione. Funziona su tutti i maggiori sistemi operativi, compreso Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64) e Windows. In questo progetto è stata utilizzata la versione 8.4 per Windows. PostgreSQL è completamente conforme al modello ACID (Atomicity, Consistency, Isolation, e Durability) dei database, fornisce supporto completo per le chiavi esterne, join, viste, triggers e stored procedures e comprende inoltre i data types più utilizzati (integer, boolean, varchar, timestamp ecc…).
Un'altra caratteristica di PostgreSQL è quella di consentire ai linguaggi di programmazione di collegarsi ad esso attraverso dei connettori, in questo modo sarà possibile definire ed eseguire le query SQL direttamente nel codice dell'applicazione. Il connettore utilizzato in questo progetto è JDBC (Java DataBase Connectivity), illustrato nel paragrafo *Librerie Java utilizzate*.

4.2 Web server: Apache Tomcat

Per la pubblicazione dell'applicazione e dei servizi web si è scelto di utilizzare Apache Tomcat [7], uno dei più diffusi web server comprendente un servlet container, che permette quindi l'esecuzione sia di servizi sia di servlet Java.

4.3 Software utilizzati

Il linguaggio di programmazione utilizzato per realizzare la parte applicativa del progetto è stato Java (vedi approfondimento paragrafo *Linguaggi di programmazione utilizzati*).
La piattaforma utilizzata è Eclipse, un ambiente di sviluppo integrato di tipo open source composta da una struttura estendibile, strumenti e runtime per la creazione, la distribuzione e la gestione del software in tutto il suo ciclo di vita. Il progetto Eclipse [8] è stato

originariamente creato da IBM nel novembre 2001 ed è sostenuto da un consorzio di fornitori di software, Eclipse Foundation, creato nel gennaio 2004 come associazione no-profit che coordina e amministra la comunità di Eclipse.
La versione utilizzata è stata la "Java EE Developers", la quale è appositamente studiata per lo sviluppo di web application Java. Per il controllo di versione del codice è stato utilizzato un plugin per Eclipse; Subeclipse [5], che implementa il sistema di controllo di versione Subversion (conosciuto anche come "svn").
Per la gestione del DBMS PostgreSQL è stato usato il software pgAdmin [6] che fornisce un'interfaccia grafica con la quale è possibile amministrare PostgreSQL in modo semplificato.
La progettazione delle strutture del database, diagrammi logici e concettuali sono state implementate tramite l'utilizzo di Microsoft Visio ®.

4.4 Linguaggi di programmazione e design pattern

4.4.1 Java

Per lo sviluppo del codice si è scelto di utilizzare Java, uno dei più famosi linguaggi di programmazione, sviluppato dalla Sun Microsystem nel 1991 e tuttora aggiornato.
Java [7] è un linguaggio interpretato, ovvero una volta eseguita la compilazione del codice si ottiene un codice intermedio, il byte code, che è eseguibile su qualsiasi sistema per cui sia disponibile l'interprete. Ciò fa di Java un linguaggio altamente portabile.
Java è anche un linguaggio di programmazione ad oggetti: un programma scritto in Java sarà composto da oggetti, ciascuno dei quali possiede determinate proprietà e prevede funzioni specifiche che gli oggetti possono svolgere. Un oggetto non deve mai elaborare direttamente i dati interni di un altro oggetto e nemmeno esporre i dati in modo da renderli accessibili ad altri oggetti. Questo concetto prende il nome di incapsulamento e consente di ottimizzare la riusabilità del codice e protegge i dati di un oggetto da cambiamenti che si produrrebbero nel caso il funzionamento del programma fosse difettoso. Riusabilità del codice e sicurezza sono stati fattori determinanti nella scelta di Java per questo progetto.

Librerie Java utilizzate

JDBC

Per accedere al database dall'applicazione è necessario utilizzare un driver per la connessione. La scelta obbligata è stata JDBC (Java DataBase Connectivity) [9], driver appositamente sviluppato per Java.
In particolare è stata utilizzata la versione implementata specificamente per PostgreSQL.
Tramite il driver è possibile effettuare tutte le operazioni SQL di base come connessioni al database, query di estrazione dati e operazioni di manipolazione dei dati, istruzioni DDL.

JDOM

Per la gestione dei file di configurazione dell'applicazione (ad esempio per impostare i parametri per la connessione al GPDW) si è scelto di utilizzare il linguaggio XML (eXtensible Markup Language) linguaggio basato su tag per la definizione di dati.
Per gestire i file XML da Java si è scelto di usare la libreria JDOM [10], che implementa l'interfaccia DOM (Document Object Model) per la manipolazione di XML. Tramite JDOM viene effettuato il parsing del documento XML e viene generato l'albero corrispondente per una facile gestione degli elementi contenuti.

JSON

JSON (JavaScript Object Notation) [11] è un formato utilizzato per lo scambio dei dati in applicazioni client-server.
La web application realizzata in questa Tesi prevede una forte interazione client-server, sia in fase di caricamento di ogni pagina, sia durante l'interazione con l'utente. Per rendere questa interazione il più confortevole possibile per l'utente, sono state implementate numerose chiamate AJAX tramite le quali vengono eseguite delle Servlet in maniera invisibile all'utente. I dati ottenuti dalle servlet vengono comunicati alle pagine web tramite JSON, che permette di trasmettere numerosi dati in maniera semplice e ordinata.

Log4J

Log4J [12] è una libreria Java sviluppata dalla Apache Software Foundation che implementa un veloce ed affidabile framework per il logging su applicazioni Java. I messaggi di log possono essere scritti su file e su console Tomcat, e possono essere dei seguenti tipi:

- ***FATAL***: segnala un errore che comporta la chiusura dell'applicazione;
- ***ERROR***: questo messaggio segnala un errore di esecuzione;

- ***WARN***: serve per segnalare condizioni inaspettate ma che non comportano necessariamente un errore;
- ***INFO***: segnala eventi di esecuzione;
- ***DEBUG***: usato nella fase di debug del codice.

4.4.2 Model-View-Controller

Per lo sviluppo del sistema si è scelto di utilizzare una basilare architettura a tre livelli, che rispetta il paradigma MVC (Model-View-Controller).

L'idea su cui si basa il pattern Model-View-Controller [14] è quella di separare i dati dell'applicazione, la logica del programma e la presentazione dei dati in entità denominate rispettivamente Model, Controller e View.

L'uso del modello di progettazione MVC produce un'architettura di applicazione flessibile in cui possono essere fornite e facilmente modificate presentazioni multiple (View) e si possono effettuare cambiamenti nella rappresentazione fisica dei dati (Model) senza toccare nulla del codice dell'interfaccia utente. L'utente interagisce con il Controller, attraverso la View, per chiedere di compiere delle operazioni. Il Controller eseguirà le richieste dell'utente e i dati elaborati saranno quindi mostrati nella View.

Nel nostro caso il livello di dati (Model) consiste di un database server che ospita il Genomic and Proteomic Data Warehouse e un database contenente i dati degli utenti e degli accessi (GPDW_web); il livello applicativo (Controller) consiste di un web server Tomcat che ospita ed espone i servizi web; per il livello di presentazione si considera invece l'interfaccia web utilizzabile dall'utente tramite browser. In Figura 9 è illustrata l'architettura della web application.

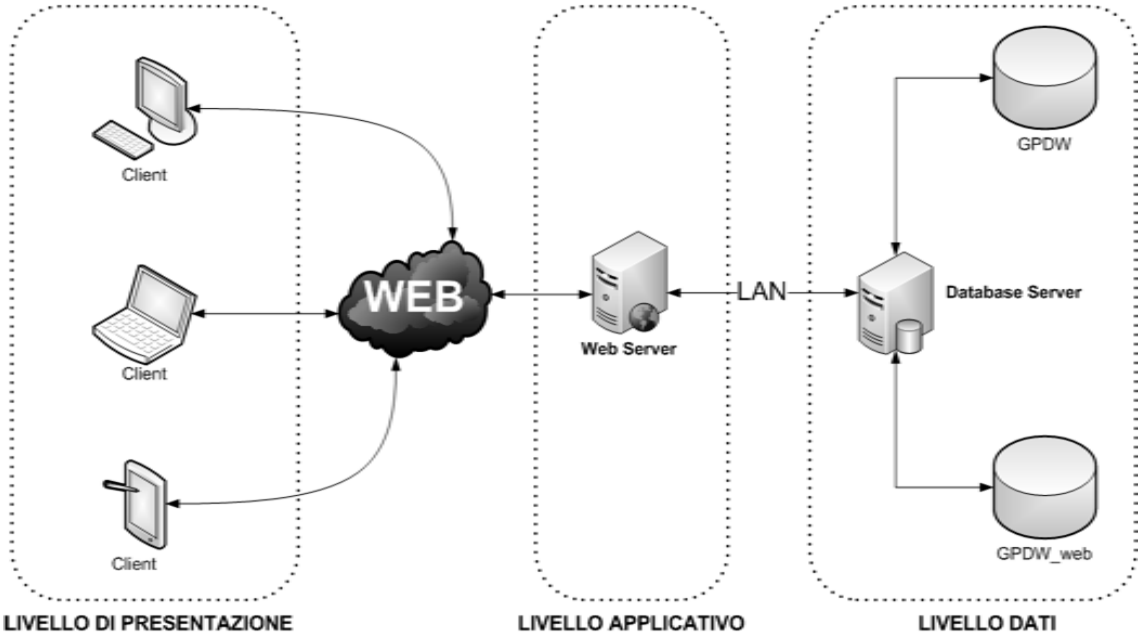


Figura 9 - Architettura a tre livelli della web application

4.5 Strumenti per la presentazione dei dati

Per la presentazione dei contenuti all'utente è stato necessario utilizzare diverse tecnologie: per i contenuti statici HTML, per quelli dinamici JSP [15] e JavaScript.

4.5.1 Lato server: JSP e Servlet

Per effettuare richieste al web server sono state utilizzate delle servlet Java [16].
Le servlet sono classi che fanno parte della piattaforma di sviluppo Java EE (Enterprise Edition); sono delle speciali classi Java che vengono eseguite su un web server che abbia anche la funzione di servlet container (nel nostro caso Tomcat). Le servlet sono classi che vengono caricate dal web server all'avvio e successivamente ricevono richieste HTTP da parte di pagine web o applicazioni; esse sono persistenti, cioè rimangono attive per più richieste. Hanno tre metodi principali:

- ***init()***: che serve a inizializzare la servlet;
- ***service()***: è il metodo utilizzato per rispondere alle richieste HTTP (e comprende i metodi doGet(), doPost(), doPut(), doDelete());
- ***destroy()***: metodo per la terminazione della servlet.
- L'utilizzo di tecnologie lato server basate su servlet comporta svariati vantaggi [12]:

- ***indipendenza da piattaforma e produttore***: essendo le servlet scritte in linguaggio Java funzionano su qualsiasi sistema operativo che supporti un ambiente di esecuzione Java;
- ***integrazione con le altre tecnologie Java*** come JDBC, utilizzato per l'accesso ai database;
- ***efficienza***: le servlet vanno in esecuzione su un processo che gira fino a quando la web application non viene terminata. Quindi tutti i dati della servlet sono salvati in memoria per tutto il tempo di esecuzione, evitando di dover ricreare un nuovo processo (e di conseguenza caricare nuovi dati) per ogni request;
- ***robustezza e sicurezza***: il linguaggio Java consente di gestire molti errori in fase di compilazione.

Per generare contenuti dinamici nelle pagine web dell'applicazione le servlet sono state utilizzate insieme alla tecnologia JSP (Java Server Pages).

Le JSP sono semplici pagine HTML contenenti dei frammenti di codice Java che consentono l'inserimento di dati dinamici in una pagina web. Le principali variabili a disposizione in una pagina JSP sono:

- ***request***: rappresenta la richiesta HTTP;
- ***response***: rappresenta la risposta HTTP;
- ***session***: rappresenta la sessione HTTP all'interno della quale è stata invocata la pagina.

4.5.2 Lato Client: HTML e CSS

Per la presentazione dei dati all'utente la scelta obbligata nell'ambito web è il linguaggio HTML (HyperText Markup Language).

Questo linguaggio permette di presentare i contenuti dell'applicazione formattati secondo uno stile definito in un file CSS (Cascading Style Sheets), che permette di gestire il posizionamento degli elementi nella pagina web.

4.5.3 Scripting dinamico: JavaScript e AJAX

Per garantire una maggiore usabilità all'applicazione si è deciso di inserire dei contenuti dinamicamente anche a pagina web già caricata: per questo è stato necessario l'utilizzo del linguaggio di scripting Javascript e di AJAX (Asynchronous JavaScript and XML), una tecnica per eseguire richieste HTTP al server in maniera asincrona. AJAX si basa sull'oggetto

XMLHttpRequest per lo scambio di dati asincrono con il web server; nonostante sia menzionato XML, può essere utilizzato qualunque formato per lo scambio di dati, come il formato JSON.
AJAX ha il vantaggio di poter generare più contenuti dinamicamente all'interno della stessa pagina web, ma ha lo svantaggio di richiedere l'attivazione di JavaScript sul client e di non essere compatibile con i browser meno recenti.

jQuery e i plug-in utilizzati

Insieme al linguaggio di scripting JavaScript si è utilizzato anche il framework jQuery.
jQuery [16]è una libreria di funzioni JavaScript che permette di semplificare la scrittura degli script e fornisce diversi metodi e funzioni per gestire al meglio aspetti grafici e strutturali di una pagina web. jQuery attualmente è un framework molto utilizzato e supporta i principali web browser (Firefox, Chrome, Opera, Safari e Internet Explorer). Le funzionalità di jQuery possono essere ampliate grazie all'utilizzo di plug-in, anche questi scritti in JavaScript.

In quesa applicazione sono stati utilizzati diversi plug-ins di jQuery.
Per la visualizzazione dei risultati in forma tabellare, è stato utilizzato il plugin ***DataTables*** [18].
In rete sono disponibili diversi plug-in appositamente realizzati per le tabelle, ma la scelta è ricaduta su DataTables in quanto tale plug-in dispone di una buona documentazione.
Le principali funzionalità di DataTables sono:

- ricerca di un termine all'interno della tabella;
- ordinamento di una o più colonne in modo crescente o decrescente;
- contenuto dinamico: i dati visualizzati sono ricavati attraverso chiamate AJAX;
- paginazione delle tabelle.

Per quanto riguarda, invece, le pagine di composizione della query, per comporre un'interfaccia dinamica e usabile sono stati utilizzati alcuni plugin per traformare i normali campi di form HTML in campi più "interattivi".
In particolare è stato utilizzato il plugin ***Multiselect***, facente parte della suite **JqueryUI** [19].
Questo plugin permette di trasofmare i normali campi select html (sia singoli che multipli) in campi più dinamici e utilizzabili.
Inoltre, per alcuni campi di tipo testo, si è voluto dotarli della feature "*autocomplete*", cioè la visualizzazione di suggerimenti durante l'inserimento dei valori da parte dell'utente. In

paricolare, è stato utilizzato il plugin di jQuery ***Tokeninput*** [20], che permette di inserire i divesi valori come "token" indivisibili all'interno del campo di testo.

4.5.4 Generazione dinamica di un grafo: SVG e Graphviz

Per quanto riguarda la generazione del grafo delle features, l'immagine finale generata è un file ***SVG*** (Scalable Vector Graphics) [21], che permettere di definire immagini in un formato XML. Questo ha permesso di poter inserire in questo file, sempre il linguaggio xml, alcune informazioni aggiuntive sul grafo non visibili nell'immagine ma comunque presenti nel file.
Per generare il file SVG del grafo è stato utilizzato ***Graphviz*** [22][23], un programma che permette di disegnare grafi scritti in linguaggio *dot*.

5 Reingegnerizzazione dell'applicazione web preesistente

In questa sezione della Tesi sono illustrate tutte le scelte progettuali effettuate e le soluzioni adottate per la reingegnerizzazione dell'applicazione web preesisente.

5.1 Salvataggio dei dati sul server

Essendo HTML un protocollo stateless, cioè tramite il quale non è possibile salvare dei dati sullo stato dell'applicazione, sono state introdotte le sessioni [16], degli strumenti residenti sul server ma associati all'istanza del browser in esecuzione sul pc dell'utente. Tramite le sessioni è possibile salvare dei dati sullo stato dell'applicazione e relativi all'utente stesso.
E' inoltre possibile salvare anche dei dati direttamente nella cache (*context*) del server. In questo modo questi dati saranno accessibili a tutti gli utenti in ogni momento in cui il server sarà in esecuzione.

5.1.1 Salvataggio dati nella sessione

Questa applicazione salva numerosi attributi all'interno della sessione, alcuni allo startup dell'applicazione, altri durante la sua esecuzione, per mantenere alcuni dati richiesti o generati dall'utente. Di seguito viene elencato il loro nome e significato.
Quando l'utente effettua il login, viene creata una sessione con i seguenti attributi:

- ***username***: il nome dell'utente
- ***user_type***: il tipo dell'utetne (amministratore, standard o avanzato)
- ***ip***, ***port***, ***dbUrl*** e ***dbName***: rispettivamente, indirizzo ip, porta, indirizzo completo e nome del database a cui si è collegati

Nel momento in cui l'utente seleziona una (o più) features su cui effettuare la ricerca, accedendo così alla pagina di composizione della query, vengono salvati i seguenti attributi:

- ***selectedFeatures***: una lista contenente i nomi delle features selezionate dall'utente

- ***selectedAssociationTables***: una lista contenente tutti i nomi delle tabelle di associazione coinvolte tra la features selezionate

Quando l'utente manda in esecuzione la query, vengono aggiornati i due attributi precedenti e salvati i seguenti dati:

- ***loadingQuery***: tutti i dati riguardanti la query composta (vedi sezione 6.2) per permettere il suo salvataggio o modifica nelle pagine successive
- ***numRecords***: il numero di records per pagina selezionati dall'utente
- ***count***: la modalità di conteggio scelta dall'utente
- ***showQuery***: il testo della query generata
- ***countQuery***: il testo della query di conteggio generata
- ***counting***: attributo che indica se è in esecuzione la query di conteggio oppure no
- ***countResult***: il risultato della query di conteggio (esatto)
- ***WhereField***: il contenuto della clausola where eventualmente generata se l'utente ha inserito dei filtri in fase di visualizzazione dei risultati

Inoltre, quando vengono eseguite la query principale (quella generata dall'utente), la query di conteggio dei risultati o una query di ricerca generata dai campi auto complete, i loro Statement vengono salvati nella sessione per poter eventualmente fermare l'esecuzione di queste query nel caso in cui l'utente lasci la pagina dei risultati oppure esegua una nuova query in un altro campo di auto complete.

Tutti i dati aggiunti dalla selezione della feature in poi, quando l'utente torna alla pagina di selezione feature vengono rimossi dalla sessione, per poter cominciare la generazione di una nuova query da una situazione "pulita".

5.1.2 Salvataggio dati nella cache del server

Oltre alla sessione, l'applicazione durante la sua esecuzione salva dei dati anche nella cache (*context*) del server. Questi dati sono accessibili a tutti gli utenti durante tutto il tempo in cui il server rimane in esecuzione.
In particolare, l'applicazione salva nella cache tutti i dati riguardanti i ***campi codificati*** dell'applicazione. Quando viene richiesta una pagina di composizione della query contenente degli attributi i cui valori sono dei campi codificati, per ognuno di questi attributi viene salvata

nella cache del server una *HashMap* il cui nome è *<nome_attributo>_encoded*, che contiene tutti i valori codificati possibili per quel campo abbinati al suo valore non codificato.
Ciò avviene perché la query di ricerca dei valori dei campi codificati si è dimostrata onerosa in termini di tempo, e viene effettuata molte volte all'interno della generazione della pagina di composizione. Ricercare invece dei termini nella cache del server risulta molto più rapido. Ciò ha permesso di abbassare di molto i tempi di generazione della pagina di composizione della query.
Per evitare di saturare la cache del server, nel caso in cui il numero dei valori di un campo codificato sia maggiore di una certa soglia (definita nel file *settings.xml*, vedi sezione 5.4), questi non vengono salvati nella cache ma verranno prelevati sempre dal database.

5.2 Separazione livello "dati" da livello "applicativo"

Per una corretta e migliore gestione dei dati negli algoritmi, è stato necessario separare nettamente il livello "dati" dal livello "applicativo", creando delle classi ad hoc per l'interazione con il database e per la manipolazione dei dati presi dallo schema public.
E' stato quindi creato un nuovo package, "*database*", contenente tutte le classi che gestiscono l'interazione con il database stesso.
Si è introdotto, inoltre, un nuovo package di classi atte a modellare i dati provenienti dal database, in particolar modo le tabelle e gli attributi di queste tabelle: il package "*databaseData*".

5.2.1 Livello "dati": classi di interazione con il database

Sono state create cinque classi di gestione dell'interazione con il database, divise a seconda della categoria delle funzioni al loro interno.

- ***DBFlags***: questa classe contiene al suo interno tutte le funzioni dedicate alla gestione dei campi codificati del database. Queste funzioni utilizzano lo schema "*flag*" del GPDW.
- ***DBMetadataGraph***: questa classe contiene tutte le funzioni utilizzate per prelevare dati dallo schema dei metadati del database e necessarie per la generazione del grafo delle features, utilizzato nella modalità grafica di composizione della query.
- ***DBQueryManager***: qui sono definite tutte le funzioni che si occupano del salvataggio, caricamento e aggiornamento delle query degli utenti salvate nel database.

- ***DBGetters***: questa classe contiene la maggior parte delle funzioni che eseguono delle query al database. Queste funzioni si occupano di prelevare dal database i dati per generare la pagina di composizione della query e altri dati utilizzati in fase di visualizzazione.
- ***DBTableGetters***: in questa classe sono presenti tutte le funzioni dedicate al prelevamento dei dati delle tabelle dello schema public nel database, e della loro rappresentazione tramite le classi del livello "applicativo" spiegate nel paragrafo seguente.

5.2.2 Livello "applicativo": package databaseData

Per gestire il livello "applicativo" dell'applicazione, è stato creato il package databaseData, di cui si riporta in Figura 10 il diagramma UML.

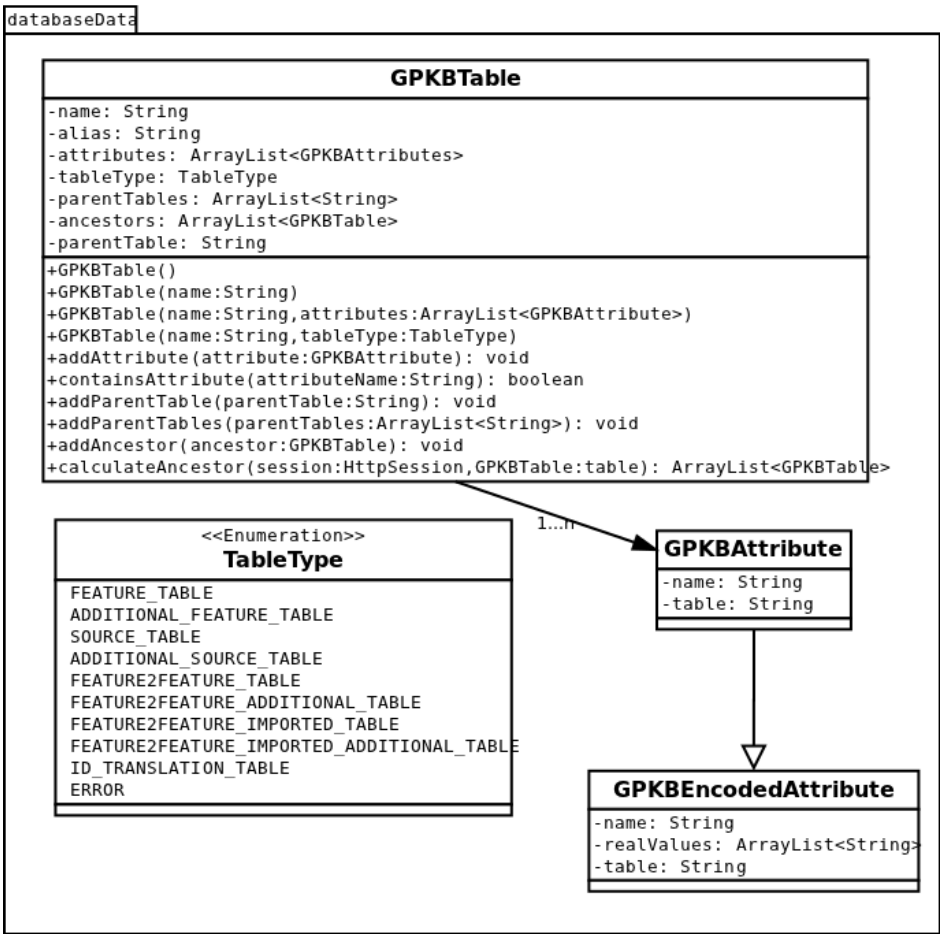


Figura 10 - UML class diagram del package databaseData

La classe principale di questo package è la classe ***GPKBTable***, che rappresenta una struttura dati atta a modellare l'elemento "tabella" prelevato dal database. Di seguito si riporta una breve spiegazione dei suoi attributi e metodi:

- ***name***: il nome della tabella.
- ***alias***: nal caso di multipla feature con feature omonime, questo attributo contiene l'alias della tabella che la distingue dalle altre, omonime.
- ***Attributes***: una lista di GPKBAttributes che rappresentano gli attributi di questa tabella
- ***tableType***: il tipo della tabella, con valori presi dall'enumerazione TableType
- ***parentTables***: una lista contenente i nomi delle tabelle padre di questa tabella
- ***ancestors***: una lista contenente le tabelle "antenato" di questa tabella
- ***parentTable***: nel caso di singolo genitore, contiene il nome della tabella padre

La tabella contiene diversi costruttori, a seconda di come è necessario inizializzarla. Contiene inoltre tutti i *getters*, i *setters* e alcuni metodi per *aggiungere elementi* alle sue liste. Il metodo ***calculateAncestors*** è un metodo statico che permette di calcolare gli antenati di una tabella (vedi sezione 6.1.1).

La classe ***GPKBAttribute***, invece, modella gli attributi di ogni tabella, contenendo, per ogni attributo, il suo ***nome*** e la sua ***tabella***. La classe ***GPKBEncodedAttribute*** estende la precedente modellando gli attributi codificati, per cui a ogni valore codificato corrisponde un valore non codificato nelle tabelle di flag. La lista ***realValues*** contiene tutti i valori non codificati possibili per quell'attributo.

5.3 Connessione al database

Come già nella precedente applicazione web, anche durante questo lavoro le informazioni necessarie alla connessione del database non sono state inserite all'interno del codice ma sono state divise da esso utilizzando un file XML di configurazione.

Rispetto alla precedente applicazione, però, è stata utilizzata una nuova libreria di connessione al database. Questo ha richiesto un cambiamento nel file di configurazione.

5.3.1 Configurazione della connessione

La connessione viene configurata impostandone tutti gli attributi nel file ***db_config.xml***.

In Tabella 2 è mostrata la parte di configurazione relativa al database di gestione degli utenti GPDW_web.

```
<database handle="gpdw_web" name="GPDW_web" checking="true">
     <description>new database for GPDW </description>
     <driver>org.postgresql.Driver</driver>
     <url_protocol>jdbc:postgresql:</url_protocol>
     <host>localhost</host>
     <port>5432</port>
     <dbms_maintenance_db>postgres</dbms_maintenance_db>
     <login>
           <user>user</user>
           <password>password</password>
     </login>
     <error_message>GPDW_web database doesn't exist</error_message>
</database>
```

Tabella 2 - Configurazione della connessione al database GPDW_web

Il valore dell'attributo *handle* del tag database rappresenta il riferimento tramite il quale, nel codice, ci si collega a questo database. Il valore dell'attributo *name*, invece, è il nome del database a cui ci si vuole collegare. I campi contenuti nel tag database indicano i dati necessari per la connessione al server di PostgreSQL..

In Tabella 3, invece, è mostrata la parte di configurazione relativa al GPDW.

```
<database handle="gpdw_system" name="GPDW_system" checking="true">
     <description>Application database</description>
     <driver>org.postgresql.Driver</driver>
     <url_protocol>jdbc:postgresql:</url_protocol>
     <host>192.168.0.1</host>
     <port>5432</port>
     <dbms_maintenance_db>postgres</dbms_maintenance_db>
     <login>
           <user>user</user>
           <password>password</password>
     </login>
     <error_message>GPDW_system database doesn't
exist</error_message>
     <datasource_list>
           <datasource handle="gpdw" sql_query="SELECT database_name
AS database_name, database_url AS database_url FROM database WHERE
database_xml_id = 'gpdw' AND is_ready = 'true' AND
switch_over_completed = 'true' ORDER BY version_date DESC LIMIT 1">
                 <description>Genomic and Proteomic Knowledge
Base</description>
                 <driver>org.postgresql.Driver</driver>
                 <dbms_maintenance_db>postgres</dbms_maintenance_db>
                 <login>
                       <user>user</user>
                       <password>password</password>
                 </login>
                 <error_message>Datasource doesn't
exist</error_message>
           </datasource>
     </datasource_list>
</database>
```

Tabella 3 - Configurazione della connessione al GPDW

In questo caso, diversamente dalla connessione illustrata precedentemente, il database finale da cui vengono estratti i dati non è il database a cui ci si collega tramite il primo *handle* (*gpdw_system*), bensì è quello indicato nel tag *datasource*, cioè "*gpdw*". In questo caso il nome del database a cui collegarsi non è definito direttamente nel file xml, ma viene prelevato, tramite la query indicata nell'attributo *sql_query*, dalla tabella "*database*" del database *GPDW_system*, che contiene tutte le informazioni sulle varie versioni del data warehouse GPDW, indicando qual è la versione stabile più aggiornata.
In questo modo, quando una nuova versione del GPDW viene dichiarata stabile, l'applicazione si collegherà automaticamente a questa nuova versione senza dover modificare il file xml.

5.4 Nuove configurazioni della web application

Oltre alle possibilità di configurazione descritte nella sezione 2.2.1, sono stati introdotti alcuni nuovi file xml che permettono di definire nuovi aspetti dell'applicazione:

- ***default_fields.xml***: in questo file è possibile indicare quali campi (a seconda del tipo di tabella, di feature o di associazione) devono essere selezionati in modo predefinito quando si genera la pagina di composizione della query
- ***userType.xml***: questo file contiene tutte i tipi di utente che possono essere presenti nell'applicazione. Ciascun tipo possiede i seguenti attributi:
    - ***id***: l'id di questo tipo utente. Coincide con il dato salvato nel database;
    - ***name***: il nome del tipo di utente;
    - ***canDisableUser***: se "true", tutti gli utenti di questo tipo potranno disabilitare gli altri utenti;
    - ***canChangeUserType***: se "true", tutti gli utenti di questo tipo potranno cambiare il tipo degli altri utenti.

  In questo momento, sono definiti tre tipi di utente: amministratore, avanzato e standard. L'unico che può disabilitare o cambiare ruolo ad altri utenti è l'utente di tipo amministratore.
- ***settings.xml***: in questo file è possibile indicare eventuali altre configurazioni. Ora sono presenti i seguenti campi:
- ***count-timeout***: indica la soglia di tempo (in millisecondi) dopo la quale interrompere la query di conteggio in modalità "*estimated*" (vedi sezione 6.2.2)
- ***encoded-fields-threshold***: indica il numero di valori dei campi codificati dopo il quale i valori di questo campo non vengono salvati nella cache del server (vedi sezione 5.1.2)

Come già spiegato nella sezione 2.2, anche in questo caso ogni file dispone di una classe adibita al recupero delle informazioni contenute in esso.

Queste classi, tutte contenute nel package "xml", sono:

- xml.DefaultFieldsXML che si occupa del file default_fields.xml
- xml.UserTypeXML che si occupa del file userType.xml
- xml.SettingsXML che si occupa del file settings.xml

6 Sviluppo dell'applicazione web

In questo capitolo verranno illustrate le tecniche utilizzate per lo sviluppo dell'applicazione web realizzata in questo lavoro di Tesi. In particolare, verranno descritti, nell'ordine, la progettazione dell'algoritmo di creazione dinamica della query di estrazione dei dati, la generazione dell'interfaccia di composizione della query (visuale e grafica), i procedimenti di analisi e visualizzazione dei risultati e la progettazione del sistema di salvataggio e caricamento delle query da parte degli utenti.

6.1 Creazione dinamica della query di estrazione dati

Nell'applicazione la generazione della query, dati tutti gli attributi e le tabelle selezionate dall'utente, si svolge in due fasi, e avviene durante l'esecuzione della servlet *ComposeQuery*.
Tutte le procedure qui descritte sono state progettate e implementate facendo costante riferimento alla struttura gerarchica e modulare del GPDW, descritta nella sezione 2.1.

6.1.1 Prima fase: generazione della lista delle tabelle completa e ordinata

La prima fase consiste nel generare una lista di tabelle che verranno inserite nella query, che sia completa e ordinata. Di questo si occupa la funzione *populateMainTables().*
Per completa si intende che contenga al suo interno tutte le tabelle da coinvolgere nei join, indipendentemente dal fatto che l'utente le abbia selezionate tutte o no.
Per controllare che una lista sia completa oppure no, si è scelto di rappresentare la struttura delle diverse tabelle con un **DAG** (**Direct Acyclic Graph**).
Come già descritto nella sezione 2.1, il GPDW è diviso in moduli, e ciascun modulo corrisponde a una feature. Ogni modulo possiede una serie di tabelle, ordinate da una gerarchia ben definita. Anche le associazioni tra moduli fanno parte di questa gerarchia.
Ogni tabella, quindi, può avere degli "antenati" (tabelle più in "alto" nel livello gerarchico) e può avere dei discendenti (tabelle più in "basso", definite "secondarie", nel livello gerarchico) (vedi Figura 4 e Figura 6). Come già indicato precedentemente, si è scelto di rappresentare

queste gerarchie con dei DAG. In questi grafi, ogni tabella è collegata a ogni suo antenato e a ogni suo discendente.

Ogni tipo di tabella, quindi, ha un suo "padre":

- Tabella di **feature**: nel caso di ricerca su singola feature, il padre è la tabella di *id_translation*. Nel caso di multipla feaure, questa tabella avrà "due padri": la tabella di *id_translation* e la (o le) tabella(e) di *associazione* con la(le) altra feature a lei associata(e).
- Tabella **addizionale di feature**: il padre è la tabella di *feature*
- Tabella di **sorgente**: il padre è la tabella di *feature*
- Tabella **addizionale di sorgente**: il padre è la tabella di *sorgente*
- Tabella di **id_translation**: il padre è la tabella di *associazione* nel caso di multipla feature, oppure nessuno *(null)*.
- Tabella di **associazione**: queste tabelle non possiedono tabelle padre
- Tabella **addizionale di associazione**: il padre è a tabella di *associazione* corrispondente
- Tabella di **associazione importata**: il padre è la tabella di *associazione* corrispondente
- Tabella **addizionale di associazione importata**: il padre è la tabella di *associazione importata* corrispondente

L'algoritmo genera tutti gli antenati di ogni tabella (*GPKBTable.calculateAncestors()*). In Figura 11 è riportato il flow chart di questo algoritmo, che implementa una funzione ricorsiva.

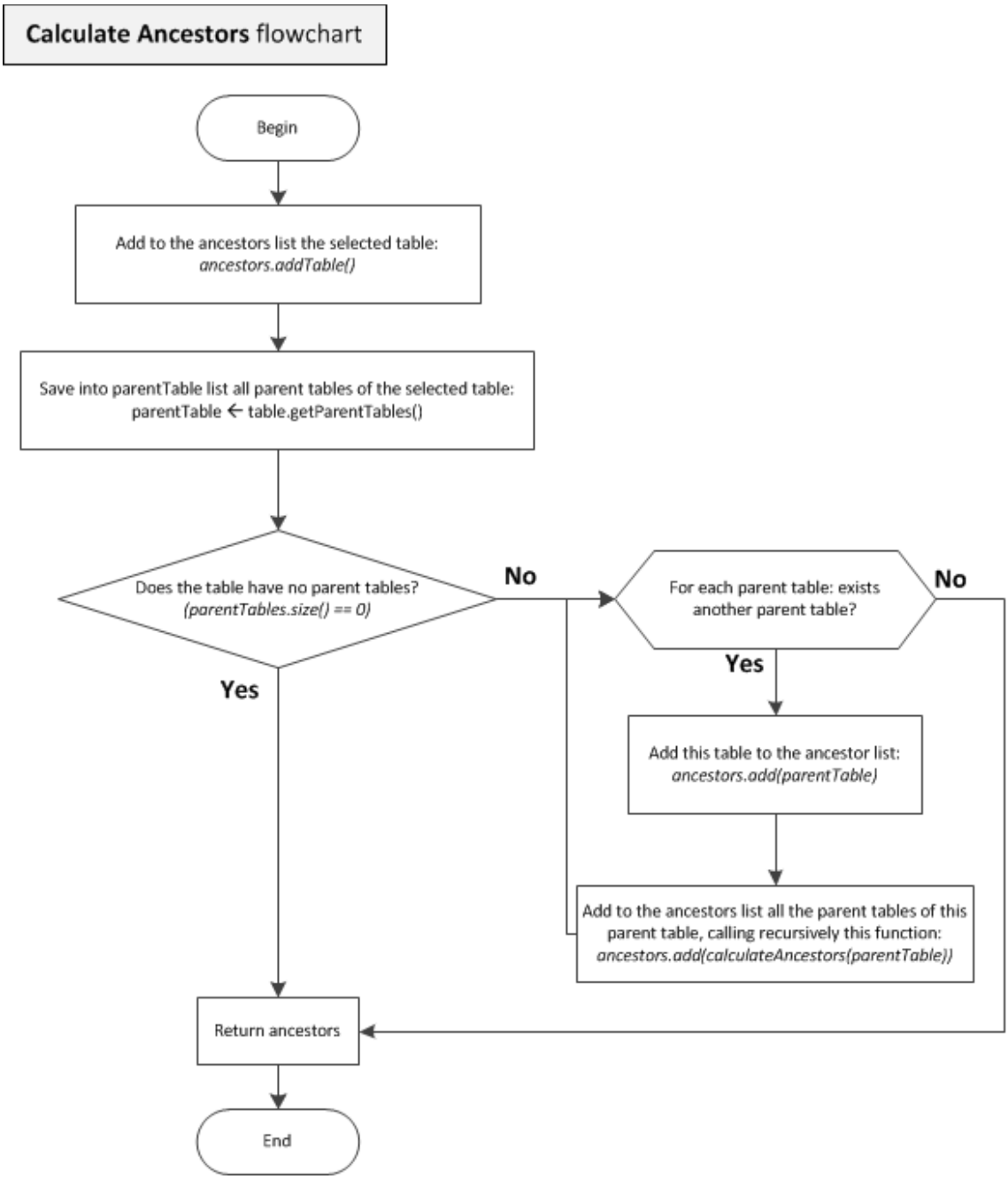


Figura 11 - Flow chart della funzione ricorsiva calculateAncestors(), eseguita per ogni tabella selezionata dall'utente. Questa funzione calcola tutti gli antenati della tabella in esame.

Dopodichè analizza il DAG creato ed estrae l'antenato comune più vicino (***lowest common ancestor***) a tutte le tabelle selezionate dall'utente, che, se non già presente, verrà incluso nella lista delle tabelle, rendendola così completa come necessario. In Figura 12 è mostrato il flowchart di questa procedura.

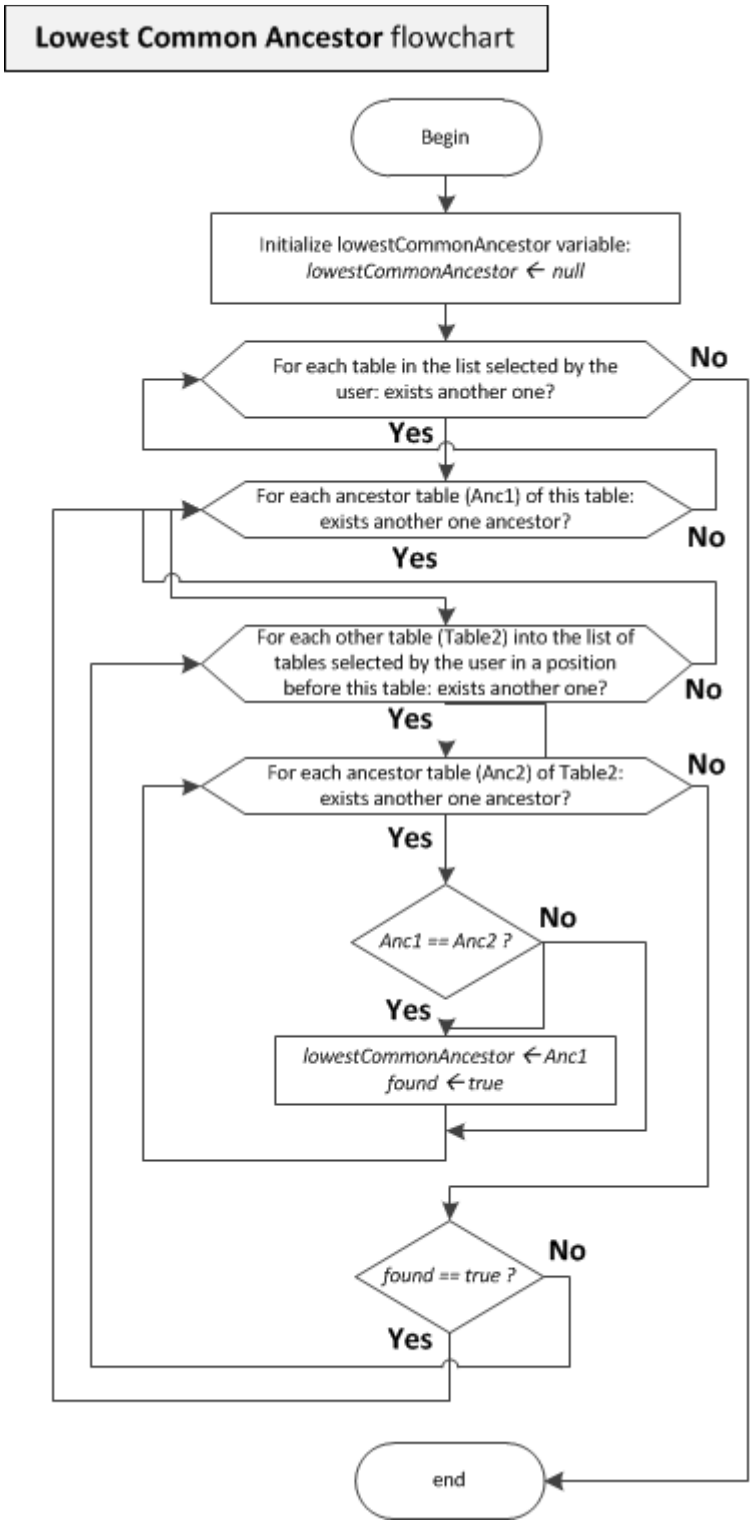


Figura 12 - Flow chart della ricerca del lowestCommonAncestors, data la lista degli antenati per ogni tabella

Se la tabella rappresentante il *lowest common ancestor* risulta già presente nella lista (perchè selezionata dall'utente), non viene aggiunta nessuna tabella.

Una volta che la lista è completa, diventa necessario ordinarla per poterla passare alla seconda parte dell'algoritmo di generazione della query. La funzione *populateMainTables()*, si occupa di ordinare la lista in questo modo:

- A feature table
- A id_translation table
- A feature additional tables
- A source tables
- A source additional tables
- A2B association table
- A2B additional association tables
- A2B imported association table
- A2B imported additional association tables
- B feature table
- B id_translation table
- B feature additional tables
- B source tables
- B source additional tables

L'ordinamento delle due feature A e B viene determinato nel modo descritto di seguito.
Date le due feature coinvolte nella query, si ricerca nella tabella del database *metadata.feature_association* (vedi sezione 2.1) la riga che identifica la loro associazione. La feature corrispondente al campo *feature1_id* sarà la feature A, la feature corrispondente al campo *feature2_id* sarà la feature B (vedi *DBGetters.getFeaturesOrder()*).
Nel caso di singola feature, l'ordine rimane lo stesso, senza le tabelle di feature B e le tabelle di associazione:

- feature table
- id_translation table
- feature additional tables
- source tables
- source additional tables

Nel caso in cui nella lista non sia presente una delle due tabelle di feature, le sue tabelle addizionali e di sorgente vengono poste in fondo alla lista, di modo che verrà fatto il join tra loro e la tabella di associazione, mentre le tabelle dell'altra feature verranno messe in cima alla lista e il loro join verrà fatto con la loro tabella di feature.
Per esempio, se mancasse la tabella di feature A, la lista sarebbe la seguente:

- B feature table
- B id_translation table
- B feature additional tables
- B source tables
- B source additional tables

- A2B association table
- A2B additional association tables
- A2B imported association table
- A2B imported additional association tables
- A id_translation table
- A feature additional tables
- A source tables
- A source additional tables

Nel caso in cui mancassero entrambe le tabelle di feature, in cima alla lista ci sarebbe la tabella di associazione cui seguirebbero poi tutte le altre, la quali faranno join solo con lei (in quanto *lowest common ancestor*).

In questo caso, la lista sarebbe la seguente:

- A2B association table
- A2B additional association tables
- A2B imported association table
- A2B imported additional association tables
- A id_translation table
- A feature additional tables
- A source tables
- A source additional tables
- B id_translation table
- B feature additional tables
- B source tables
- B source additional tables

L'algoritmo di ordinamento funziona in questo modo: dalla lista delle tabelle selezionate dall'utente (a cui è stata aggiunta la tabella rappresentante il *lowest common ancestor*), viene creata una nuova lista.

Prima vengono aggiunte le tabelle di associazione presenti nella vecchia lista. Quindi, per ognuna di queste tabelle di associazione, se presenti nella vecchia lista, vengono agguinte le tabelle di feature che questa tabella associa, nell'ordine dato dalla tabella di associazione. Ad esempio:

- Tabelle presenti nella vecchia lista:
- protein
- gene
- gene2protein
- Tabelle inserite nella nuova lista, in ordine:
- **gene** (feature A)
- gene2protein (A2B)
- **protein** (feature B)

Di seguito, poi, se selezionata dall'utente, dopo la relativa tabella di feature viene aggiunta la tabella di id_translation. Infine, ogni tabella rimanente viene posta dopo il suo antenato più vicino già presente nella nuova lista.

Nel caso di multiple feature search, il procedimento di generazione della lista rimane invariato, con l'aggiunta delle nuove tabelle riguardanti le feature aggiuntive. A titolo di esempio, si riportano le liste generate nel caso di tre feature.

Lista completa:

- A feature table
- A id_translation table
- A feature additional tables
- A source tables
- A source additional tables
- A2B association table
- A2B additional association tables
- A2B imported association table
- A2B imported additional association tables
- B feature table
- B id_translation table
- B feature additional tables
- B source tables
- B source additional tables
- B2C association table
- B2C additional association tables
- B2C imported association table
- B2C imported additional association tables
- C feature table
- C id_translation table
- C feature additional tables
- C source tables
- C source additional tables

Lista senza feature tables:

- A2B association table
- A2B additional association tables
- A2B imported association table
- A2B imported additional association tables
- A id_translation table
- A feature additional tables
- A source tables
- A source additional tables
- B id_translation table
- B feature additional tables
- B source tables
- B source additional tables
- B2C association table

- B2C additional association tables
- B2C imported association table
- B2C imported additional association tables
- C id_translation table
- C feature additional tables
- C source tables
- C source additional tables

6.1.2 Seconda fase: generazione dei join

Una volta generata la lista completa e ordinata, la servlet *ComposeQuery* si occupa di passare in rassegna gli elementi di questa lista e generare i join tra le tabelle.

L'algoritmo di generazione di join procede in questo modo: scorre la lista ordinata delle tabelle e, quando incontra una tabella principale (cioè di feature o di associazione, vedi sezione 2.1), la "salva" come tabella con cui le tabelle successive dovranno fare il join. (Di seguito si riportano diversi esempi concreti).

Quindi, nel caso in cui nella lista siano presenti entrambe le tabelle di feature A e B e la tabella di associazione A2B, prima vengono inseriti tutti i join tra la feature A e le sue tabelle, poi i join tra la tabella di associazione e le sue tabelle, e poi i join tra la feature B e le sue tabelle.
Nel caso in cui si consideri una tabella secondaria (cioè che non sia di feature o di associazione), prima di effettuare il join tra lei e la sua tabella principale relativa, vengono passate in rassegna tutte le tabelle a lei precedenti nella lista e viene controllato se esiste un vincolo di chiave esterna tra la tabella in questione e una di quelle a lei precedenti nella lista. Se così è, viene inserito il join tra queste due tabelle e non con la tabella principale. Questo procedimento viene effettuato per motivi di ottimizzazione: la query sarebbe corretta effettuando il join anche direttamente con la tabella principale. Per come è strutturato il GPDW, però, le tabelle secondarie possiedono meno tuple rispetto alle tabelle principali. Di conseguenza, un join tra due tabelle secondarie risulta essere più veloce rispetto a un join tra una tabella secondaria e una principale.

Quando si considera una tabella di associazione, vengono generati i join tra questa tabella e le due tabelle di feature a lei relative, quando presenti.
Se una di questa tabelle di feature (o entrambe) non è stata selezionata dall'utente, viene ricercata nella lista la tabella relativa alla feature mancante (cioè la prima tabella del suo

"modulo": una sua tabella di sorgente, o addizionale di sorgente, o addizionale di feature). Se non viene trovata nessuna tabella, allora viene ricercata la tabella di associazione più vicina e viene effettuato il join con questa. Se invece non c'è nessuna tabella di associazione, non viene effettuato nessun join.

Ogni volta che viene generato un join, viene scelto quale tabella includere nella clausola ON.
Il criterio è il seguente: ogni volta che viene generato un join, la tabella inserita nella clausola ON viene salvata in una lista (*tablesInJoinOn*). La prima tabella a essere inserita in questa lista sarà quella della clausola FROM (cioè la prima tabella della lista).
Quando deve essere generato un join, viene controllato se la tabella con cui fare il join (cioè la tabella non principale) è già presente nella lista *tablesInJoinOn*. Se così è, viene controllato che nella lista non sia presente la tabella principale. Se non è presente, nella clausola ON del join viene inserita la tabella principale. Altrimenti, non viene generato il join, perché significa che entrambe le tabelle sono state inserite in una clausola ON di un join precedente, e inserendole in una nuova clausola ON verrebbe generata una query sintatticamente scorretta.
In Figura 13 è riportato il flow chart dell'algoritmo di generazione dei join.

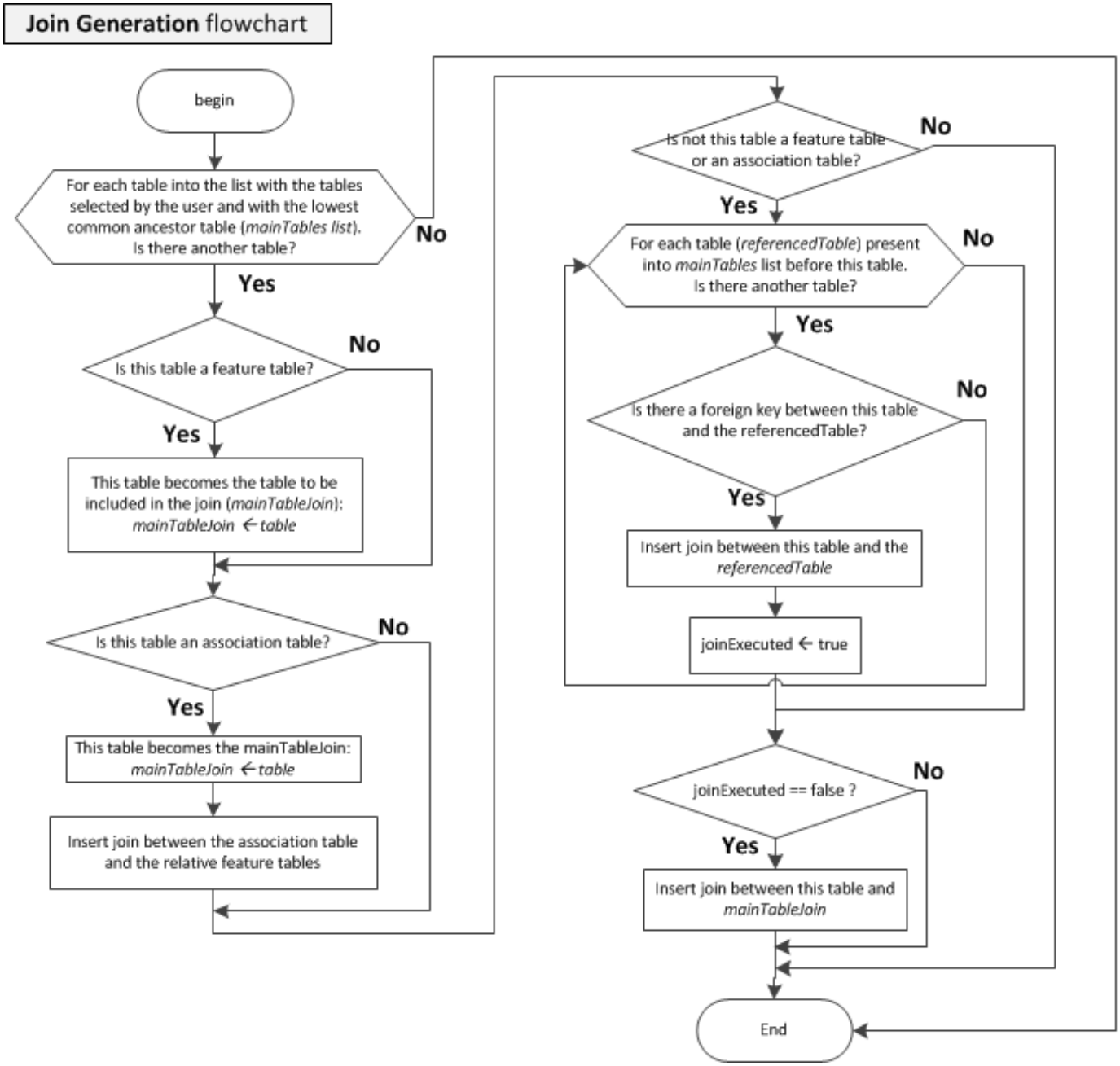


Figura 13 - Flow chart dell'algoritmo di generazione dei join tra le tabelle selezionate dall'utente

Di seguito si riporta un esempio concreto di esecuzione dell'algoritmo:

Supponiamo che la lista delle tabelle, completa e ordinata, sia la seguente:

- **gene** (feature A)
- gene_id_translation
- entrez_gene
- entrez_gene_alternate_description
- gene2biological_function_feature (A2B)
- biological_function_feature (feature B)

- biological_function_feature_id_translation
- biological_function_feature_alternative_id
- go_biological_process

Di conseguenza, l'algoritmo funziona così (con la variabile ***mainTableJoin*** si indica la tabella con la quale verranno effettuati i join):

1. Tabella: **gene** (tabella di feature). Non viene creato alcun join (la prima tabella sarà nella parte "from" della query), però ***mainTableJoin*** = "gene".
2. Tabella: *gene_id_translation* (tabella di id_translation). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista (in questo caso, solo **gene**) un vincolo di chiave esterna. Così è, quindi:
    - ***inserito il join tra gene e gene_id_translation***.
3. Tabella: *entrez_gene* (tabella di sorgente). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista (*gene* e *gene_id_translation*) un vincolo di chiave esterna. Così è con la tabella *gene*, quindi:
    - ***inserito il join tra gene e entrez_gene***.
4. Tabella: *entrez_gene_alternate_description* (tabella addizionale di sorgente). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così è con la tabella *entrez_gene* Quindi:
    - ***inserito il join tra entrez_gene e entrez_gene_alternate_description***.
5. Tabella: gene2biological_function_feature (tabella di associazione). E' una tabella principale, quindi mainTableJoin = "gene2biological_function_feature". Inoltre, è una tabella di associazione, quindi si controlla che siano presenti nella query entrambe le features e si inseriscono i join tra la tabella di associazione e le due feature. Quindi:
    - ***inserito il join tra gene e gene2biological_function_feature***
    - ***inserito il join tra biological_function_feature e gene2biological_function_feature***
6. Tabella: **biological_function_feature** (tabella di feature). E' una tabella principale, quindi ***mainTableJoin*** = "biological_function_feature". Non si inserisce nessun join, in quanto precedentemente già inserito il join con la tabella di associazione *gene2biological_function_feature*.
7. Tabella: *biological_function_feature_id_translation* (tabella di id_translation). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così è, con la tabella *biological_function_feature*. Quindi:
    - ***inserito il join tra biological_function_feature e biological_function_feature _id_translation***.

8. Tabella: *biological_function_feature_alternative_id* (tabella addizionale di feature). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così è, con la tabella *biological_function_feature*. Quindi:
    - ***inserito il join tra biological_function_feature e biological_function_feature_alternative_id***.
9. Tabella: *go_biological_process* (tabella di sorgente). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così è, con la tabella *biological_function_feature*. Quindi:
    - ***inserito il join tra biological_function_feature e go_biological_process***.

E ora la query è completa. La clausola FROM generata è mostrata nella Tabella 4.

```
FROM public.gene
INNER JOIN public.gene_id_translation ON
public.gene.gene_oid=public.gene_id_translation.translated_gene_oid
INNER JOIN public.entrez_gene ON
public.gene.gene_oid=public.entrez_gene.entrez_gene_oid
INNER JOIN public.entrez_gene_alternate_description ON
public.entrez_gene.entrez_gene_oid=
public.entrez_gene_alternate_description.entrez_gene_oid
INNER JOIN public.gene2biological_function_feature ON
public.gene.gene_oid=public.gene2biological_function_feature.gene_oid
INNER JOIN public.biological_function_feature ON
public.biological_function_feature.biological_function_feature_oid=
public.gene2biological_function_feature.biological_function_feature_oid
INNER JOIN public.biological_function_feature_id_translation ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature_id_translation.translated_biological_f
unction_feature_oid
INNER JOIN public.biological_function_feature_alternative_id ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature_alternative_id.biological_function_fea
ture_oid
INNER JOIN public.go_biological_process ON
public.biological_function_feature.biological_function_feature_oid=
public.go_biological_process.go_biological_process_oid
```

Tabella 4 - Clausola FROM geneata dall'algoritmo: primo esempio

Di seguito un secondo esempio concreto, nel caso non sia presente la tabella di feature A (ma solo la sua tabella di id_translation) e sia presente una tabella addizionale di associazione.

- **small_molec** (feature A)
- transcript2small_molec (A2B)
- confidence_4_transcript2small_molec
- transcript_id_translation

L'algoritmo funziona così (con la variabile ***mainTableJoin*** si indica la tabella con la quale verranno effettuati i join):

1. Tabella: **small_molec** (tabella di feature). Non viene creato alcun join (la prima tabella sarà nella parte "from" della query), però ***mainTableJoin*** = "small_molec".

2. Tabella: **transcript2small_molec** (tabella di associazione). E' una tabella principale, quindi ***mainTableJoin*** = "transcript2small_molec". Inoltre, è una tabella di associazione, quindi si controlla che siano presenti nella query entrambe le features e inserisco i join tra la tabella di associazione e le due feature. In questo caso è presente solo una tabella di feature. Quindi:

- ***inserito (soltanto) il join tra small_molec e transcript2small_molec***

3. Tabella: *confidence_4_transcript2small_molec* (tabella addizionale di associazione). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così è, con la tabella *transcript2small_molec*. Quindi:

- ***inserito il join tra confidence_4_transcript2small_molec e transcript2small_molec***.

4. Tabella: *transcript_id_translation* (tabella di id_translation).). Non è una tabella principale, quindi si controlla se è presente tra lei e le tabelle a lei precedenti nella lista un vincolo di chiave esterna. Così **non** è quindi aggiungo il join tra la tabella e la mainTableJoin:

- ***inserito il join tra transcript_id_translation e transcript2small_molec***.

E ora la query è completa. La clausola FROM generata è mostrata in Tabella 5.

```
FROM public.small_molec

INNER JOIN public.transcript2small_molec ON
public.small_molec.small_molec_oid =
public.transcript2small_molec.small_molec_oid

INNER JOIN public.confidence_4_transcript2small_molec ON
public.transcript2small_molec.association_oid=
public.confidence_4_transcript2small_molec.association_oid

INNER JOIN public.transcript_id_translation ON
public.transcript2small_molec.transcript_oid=
public.transcript_id_translation.translated_transcript_oid
```

Tabella 5 - Clausola FROM geneata dall'algoritmo: secondo esempio

Infine, si consideri una query *multiple feature* che viene generata selezionando dal grafo i tre nodi "*biological_function_feature*", "*enzyme*" e "*protein_fam_dom*" e, per la feature enzyme, non selezioniamo la sua tabella di feature ma solo una sua tabella di sorgente, "*expasy_enzyme*".

La lista delle tabelle completa e ordinata è la seguente:

- biological_function_feature (feature A)
- biological_function_feature2enzyme
- expasy_enzyme
- biological_function_feature2protein_fam_dom (A2C)
- protein_fam_dom (feature C)
- enzyme2protein_fam_dom (B2C)

I join verranno generati nel modo seguente:

1. Tabella: **biological_function_feature** (tabella di feature).
   Non viene creato alcun join (la prima tabella sarà nella parte "from" della query), però ***mainTableJoin*** = "biological_function_feature".
   E' la prima tabella, sarà quella inserita nella clausola FROM, quindi **tablesInJoinOn** = *"biological_function_feature"*.

2. Tabella: **biological_function_feature2enzyme** (tabella di associazione).
   E' una tabella principale, quindi ***mainTableJoin*** = "biological_function_feature2enzyme".
   Inoltre, è una tabella di associazione, quindi si controlla che siano presenti nella query entrambe le features e si inseriscono i join tra la tabella di associazione e le due feature, controllando il contenuto della lista tablesInJoinOn.
   La tabella enzyme non è contenuta nella lista. Si ricerca quindi una tabella relativa e enzyme. Trovo "*expasy_enzyme*".
   Quindi:
   - inserito il join tra **biological_function_feature** e **biological_function_feature2enzyme** (clausola ON: **biological_function_feature2enzyme**)
   - inserito il join tra **expasy_enzyme** e **biological_function_feature2enzyme** (clausola ON: **expasy_enzyme**)
   - **tablesInJoinOn** = *"biological_function_feature", "biological_function_feature2enzyme", "expasy_enzyme"*
3. Tabella: *expasy_enzyme* (tabella di sorgente).
   Non viene creato alcun join, in quanto è una tabella secondaria che non ha vincoli di chiave esterna con nessuna delle tabelle che la precedono nella lista.

4. Tabella: **biological_function_feature2protein_fam_dom**.
   E' una tabella principale, quindi ***mainTableJoin*** = "biological_function_feature2protein_fam_dom".
   Inoltre, è una tabella di associazione, quindi si controlla che siano presenti nella query entrambe le features e si inseriscono i join tra la tabella di associazione e le due feature, controllando il contenuto della lista *tablesInJoinOn*.
   Quindi:

   - inserito il join tra **biological_function_feature** e **biological_function_feature2protein_fam_dom** (clausola ON: **biological_function_feature2protein_fam_dom**)
   - inserito il join tra **protein_fam_dom** e **biological_function_feature2protein_fam_dom** (clausola ON: **protein_fam_dom**)
   - **tablesInJoinOn** = *"biological_function_feature", "biological_function_feature2enzyme", "enzyme", "biological_function_feature2protein_fam_dom", "protein_fam_dom"*

5. Tabella: **protein_fam_dom**. (tabella di feature).
   Non viene creato alcun join, in quanto già precedentemente inserito il join con la tabella di associazione *biological_function_feature2protein_fam_dom*. In ogni caso, ***mainTableJoin*** = "protein_fam_dom".

6. Tabella: **enzyme2protein_fam_dom**.
   E' una tabella principale, quindi ***mainTableJoin*** = "enzyme2protein_fam_dom".
   Inoltre, è una tabella di associazione, quindi si controlla che siano presenti nella query entrambe le features e si inseriscono i join tra la tabella di associazione e le due feature, controllando il contenuto della lista *tablesInJoinOn*.
   La tabella enzyme non è contenuta nella lista. Si ricerca quindi una tabella relativa a enzyme. Trovo "*expasy_enzyme*".
   Quindi:

   - inserito il join tra **expasy_enzyme** e **enzyme2protein_fam_dom** (clausola ON: **enzyme2protein_fam_dom**)
   - si prova a inserire il join tra **protein_fam_dom** e **enzyme2protein_fam_dom** (clausola ON: ***entrambe le tabelle sono già presenti in tablesInJoinOn. NON INSERISCO IL JOIN***)
   - **tablesInJoinOn** = *"biological_function_feature", "biological_function_feature2enzyme", "enzyme", "biological_function_feature2protein_fam_dom", "protein_fam_dom", "enzyme2protein_fam_dom"*

E ora la query è completa. La clausola FROM generata è riportata in Tabella 6.

```
FROM public.biological_function_feature

INNER JOIN public.biological_function_feature2enzyme ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature2enzyme.biological_function_feature_oid

INNER JOIN public.expasy_enzyme ON
public.expasy_enzyme.expasy_enzyme_oid=
public.biological_function_feature2enzyme.enzyme_oid

INNER JOIN public.biological_function_feature2protein_fam_dom ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature2protein_fam_dom.biological_function_fe
ature_oid

INNER JOIN public.protein_fam_dom ON
public.protein_fam_dom.protein_fam_dom_oid=
public.biological_function_feature2protein_fam_dom.protein_fam_dom_oid

INNER JOIN public.enzyme2protein_fam_dom ON
public.expasy_enzyme.expasy_enzyme_oid=
public.enzyme2protein_fam_dom.enzyme_oid
```

Tabella 6 - Clausola FROM geneata dall'algoritmo: terzo esempio

Di seguito si riporta la rappresentazione del DAG per alcune situazioni di generazione di query.

I **nodi grigi** indicano le tabelle *selezionate dall'utente*, il **nodo bianco con il bordo in grassetto** rappresenta il *lowest common ancestor*.

Gli **archi neri** sono le relazioni *padre-figlio*, gli **archi tratteggiati** indicano le *relazioni con antenati* e gli **archi in grassetto** indicano il *percorso di join* incluso nella query.

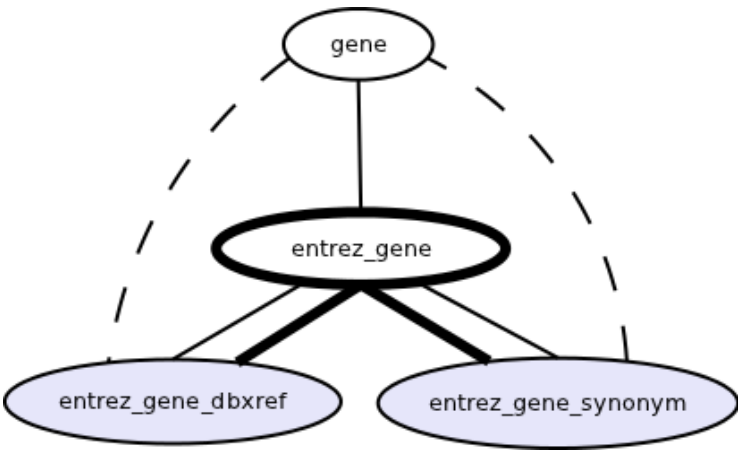


Figura 14 - Direct Acyclic Graph (DAG) singola feature: gene. Selezione di due tabelle addizionali di sorgente.

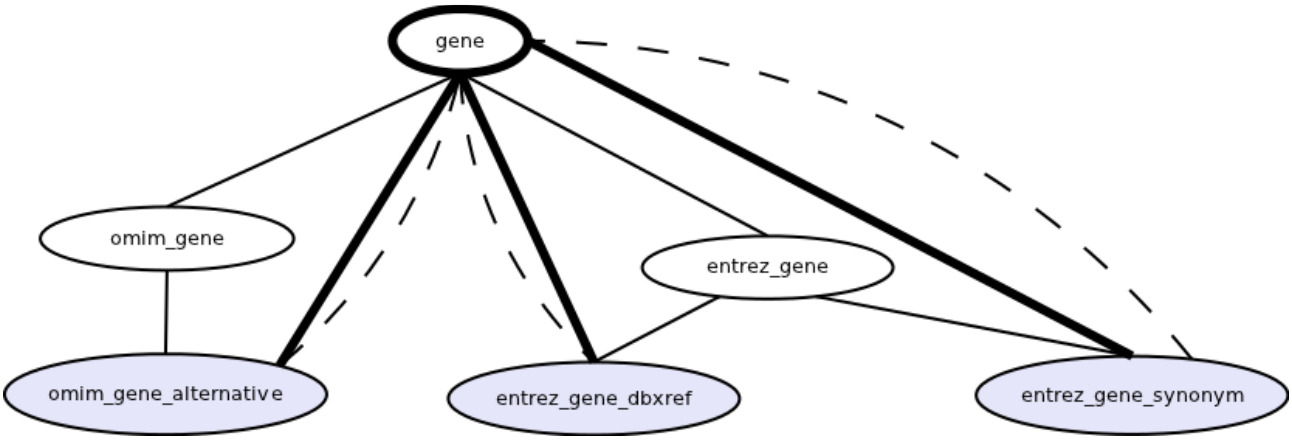


Figura 15 – DAG singola feature: gene. Selezione di tabelle addizionali di sorgente di due sorgenti differenti.

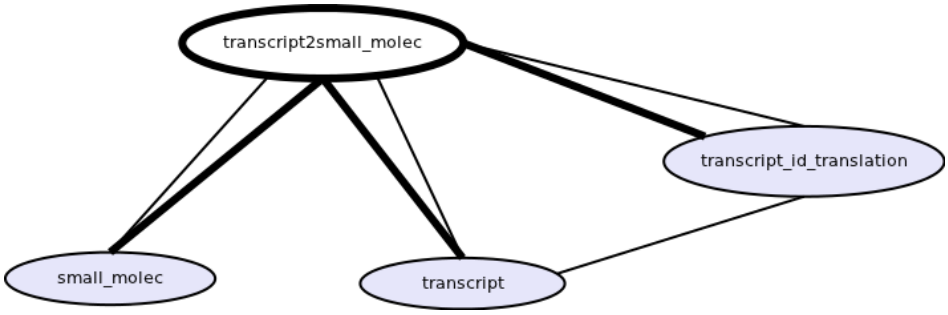


Figura 16 – DAG doppia feature: transcript e small_molec. Selezione delle tabelle di feature e della tabella di id_translation.

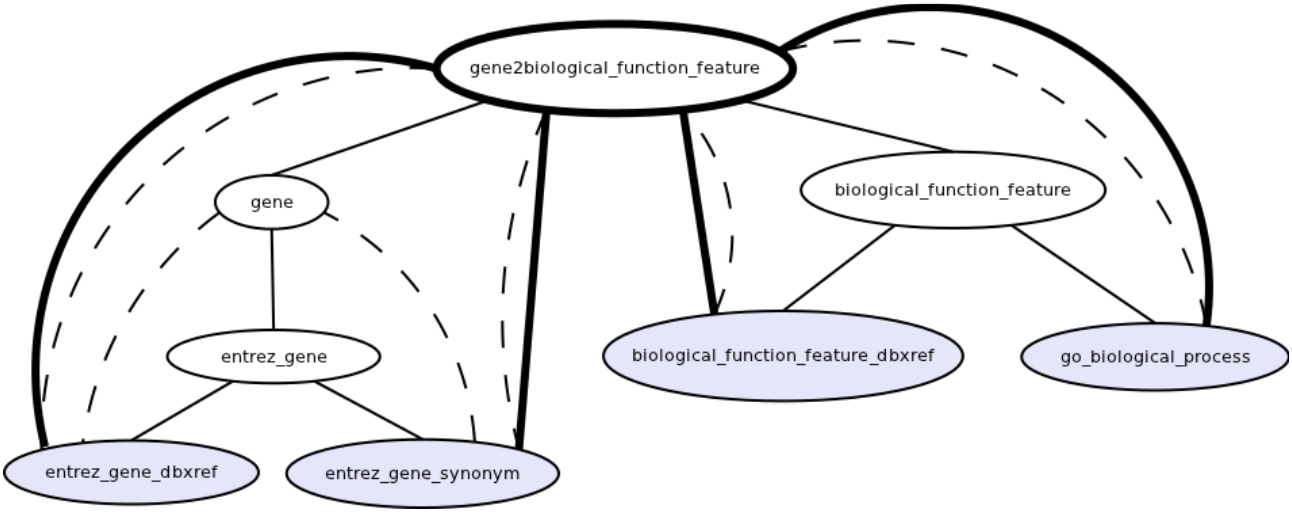


Figura 17 – DAG doppia feature: gene e biological_function_feature. Selezione di tabelle addizionali di sorgente di gene e tabelle addizionali di sorgente e addizionali di feature di biological_function_feature

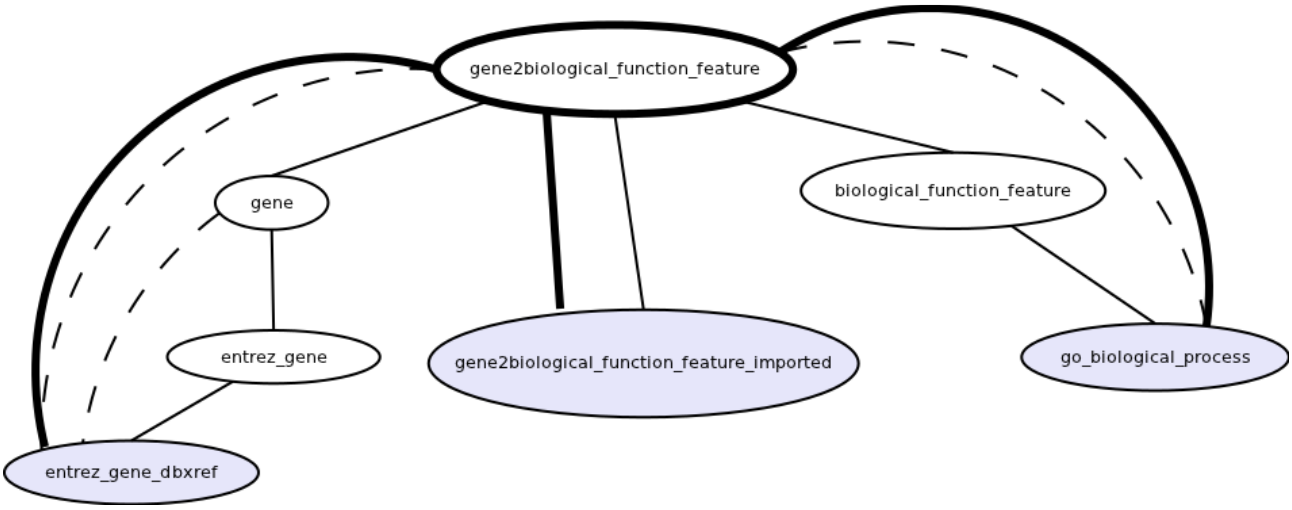


Figura 18 – DAG doppia feature: gene e biological_function_feature. Selezione di una tabella addizionale di sorgente di gene, di una tabella di sorgente di biological_function_feature e della tabella di associazione importata.

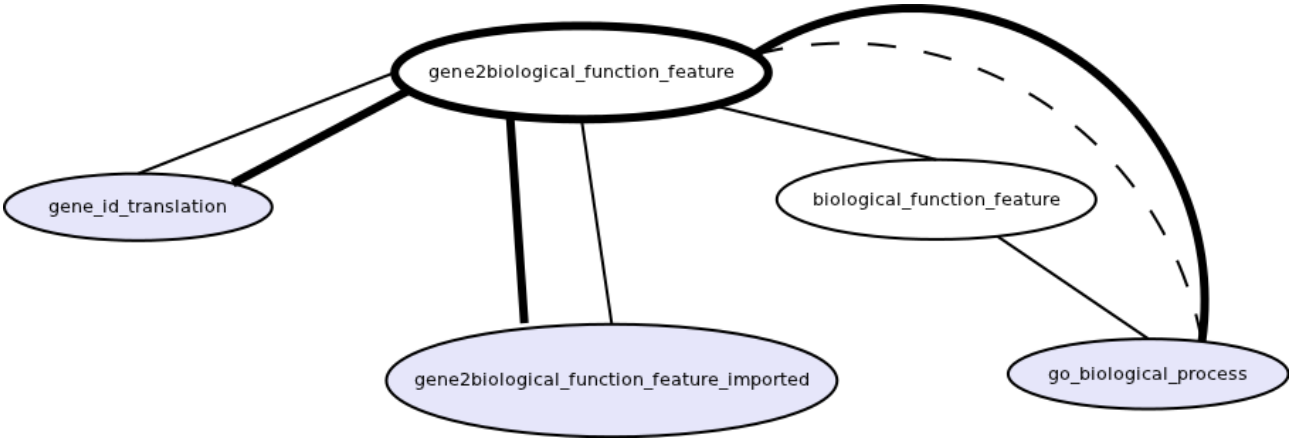


Figura 19 – DAG doppia feature: gene e biological_function_feature. Selezione di gene_id_translation. Se l'utente non seleziona la tabella gene, gene_id_translation farà il join direttamente con la tabella di associazione.

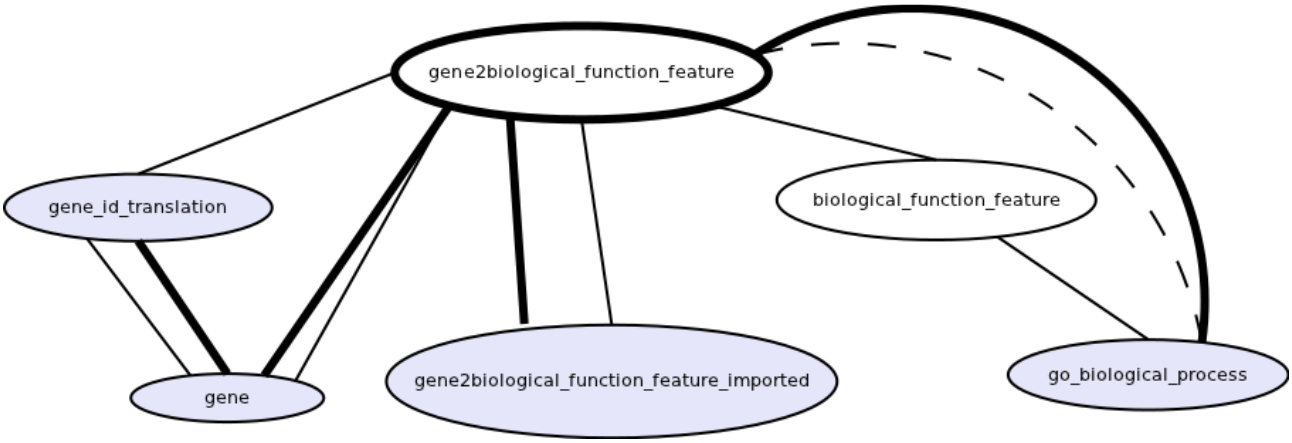


Figura 20 – DAG doppia feature: gene e biological_function_feature. Selezione di gene_id_translation. Selezionando la tabella gene, il percorso di join cambia.

Selezione degli attributi da coinvolgere nel join

Ogni tabella del GPDW possiede uno (o più) campi i cui nome termina con "_oid". Questi campi, che non vengono mostrati all'utente durante la creazione della query, sono campi di "servizio" necessari per identificare in modo univoco una tupla all'interno della sua tabella e per definire i vincoli di chiave esterna tra le tabelle. E' per questo motivo che tutti i join (ad eccezione di quelli effettuati per la query "concept", vedi sezione 6.1.4) sono svolti su attributi di questo tipo.

La generazione della clausole di join è svolta dalla classe *JoinManager*.
Il criterio di selezione dei campi delle tabelle su cui fare il join è il seguente:

- se è presente un *vincolo di chiave esterna* tra le due tabelle coinvolte nel join, si utilizzano i campi coinvolti in questo vincolo
- se non è presente un vincolo di chiave esterna e entrambe le tabelle contengono *solo un attributo che termina con "_oid"*, allora il join è svolto su questi due campi. Per esempio, è il caso di un join tra una tabella di sorgente addizionale e una tabella di feature.
- se non è presente nessun vincolo di chiave esterna e *una delle due tabelle ha più campi che terminano con "_oid"*, viene ricercato il campo corretto con cui effettuare il join, nel modo seguente.

  Sarà indicata come tabella "1" la tabella che possiede un solo campo "_oid", come tabella "2" quella che possiede più campi "_oid". Per ognuno dei campi che terminano con *"_oid"* della tabella 2, viene cercata la tabella a cui questo campo è collegato tramite un vincolo di chiave esterna. Viene così generata una lista di tabelle "collegate" (da vincoli di chiave esterna) alla tabella 2. Se non esistesse nessun vincolo di chiave esterna su questo campo "_oid", non verrebbe aggiunta alla lista nessuna tabella.

  Poi, per il campo "_oid" della tabella 1 viene ricercata nei metadati (*DBTableGetters.getLinkedFeature())* la tabella di feature a cui è collegata. Se questa tabella (che chiameremo tabella "comune") è presente nella lista delle tabelle "collegate" alla tabella 2, allora viene effettuato il join tra il campo "_oid" della tabella 1 e il campo "_oid" della tabella 2 che referenzia la tabella "comune". Se non viene trovato nessun oid corrispondente, viene generata una clausola di join vuota, e il join tra queste due tabelle non viene inserito nella query. Questa situazione, in assenza di

errori nel GPDW, non deve accadere. Viene quindi generato un warning mostrato agli utenti che li avvisa di problemi nella struttura della base di dati.

Questo caso appena descritto è il caso del join, per esempio, tra una tabella di *sorgente* e una tabella di *associazione*. La tabella di associazione ha diversi campi che terminano con "_oid": *feature_A_oid, feature_B_oid, association_oid* e *annotation_oid*
In questo caso l'algoritmo, per ognuno di questi campi, ricerca la tabella a cui è legato da un vincolo di chiave esterna (cioè la tabella di feature A per *A_feature_oid*, la tabella di feature B per *B_feature_oid* e la tabella di associazione importata per *association_oid)*.

Dopodichè, per l'altra tabella che possiede un solo campo "_oid" (nell'esempio, la tabella di sorgente), viene ricercata la tabella di feature a cui è collegata, tramite delle query ai metadati (*DBTableGetters.getLinkedFeature()*). Se questa tabella è una delle tre trovate nella ricerca precedente, allora verrà usato l'oid corrispondente.

- il caso in cui entrambe le tabelle possiedano più campi che terminano con "_oid" non è possibile che accada in condizioni di corretto funzionamento della base di dati. Nel caso in cui ci fosse un errore e si verifichi questa eventualità, viene generata una clausola di join vuota, che non viene inserita nella query, senza inficiare la correttezza sintattica della stessa. Viene poi generato un warning mostrato agli utenti che li avvisa di problemi nella struttura della base di dati.

In Figura 21 è riportato il flowchart di questo algoritmo.

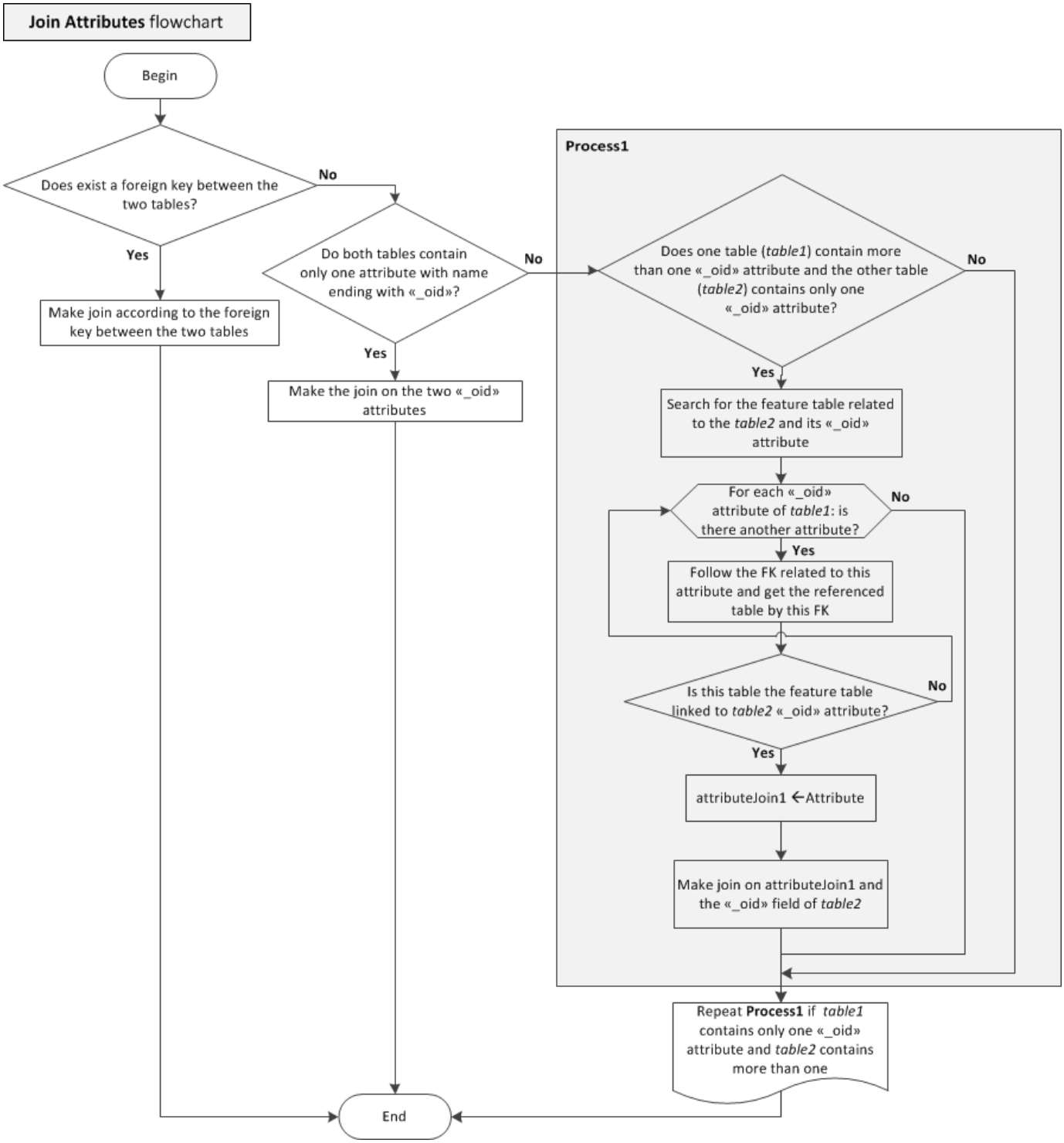


Figura 21 - Flow chart algoritmo di selezione attributi per il join

6.1.3 Definizione del tipo di "OUTER" join

Nel caso di singola feature e nel caso di feature multipla (se l'utente non ha selezionato l'opzione "*Only matching*"), tutti i join della query generata vengono generati come ***OUTER JOIN***.

Il criterio per definire se effettuare un ***LEFT*** o un ***RIGHT*** join è stabilito in base al tipo di tabelle coinvolte nel join stesso.

La regola generale si basa sul fatto di voler ottenere tutte le tuple possibile effettuando un *outer join*. Di conseguenza, sfruttando la struttura gerarchica del database GPDW (vedi sezione 2.1) e l'algoritmo precedentemente esposto, si possono definire delle regole generali perché ciò non avvenga.

Seguendo i join effettuati dall'algoritmo, nella maggior parte dei casi vengono effettuati join tra tabelle gerarchicamente più in "alto" (e quindi contenenti tutte (e più) le tuple delle tabelle dei livelli inferiori) con tabelle di livelli inferiori. Di conseguenza, nella maggior parte dei casi verrà effettuato un *LEFT* join.

Ci sono però alcuni casi particolari:

- *tabelle di id_translation*: queste tabelle sono poste dopo la tabella principale di feature nella lista delle tabelle nell'ambito di ciascuna feature, ma contengono tutte e più tuple sia della tabella principale di feature che di tutte le altre. Di conseguenza, quando in un join è presente una tabella di id_translation, questo sarà un ***RIGHT*** join.
- *tabelle di associazione importate (anche addizionali)*: analogamente, queste tabelle contengono tutte (e più) le tuple delle tabelle di associazione con cui faranno il join. Di conseguenza ogni outer join che le coinvolge sarà un ***RIGHT*** join.

6.1.4 Query "concept"

Nel database sono presenti diverse istanze di uno stesso oggetto fornite da diverse sorgenti (ad esempio diverse istanze di uno stesso gene, una fornita da *entrez_gene*, l'altra fornita da *omim*), con, eventualmente, diversi id. Ognuna di queste istanze può avere associazioni diverse con altri oggetti (della stessa o di altre features).

Eseguendo una query standard, si recuperano queste istanze come diverse, ognuna con le proprie associazioni.

Grazie a dati di *similarità* e di *history* di id contenuti nel database, in fase di creazione del database istanze diverse di uno stesso oggetto vengono identificate come riferite allo stesso oggetto ("*concept*") e si associa a tutte queste istanze lo stesso *concept_oid*.
Sfruttando ciò, è possibile recuperare tutti i dati (e associazioni) di uno stesso oggetto presenti in tutte le sue varie istanze (in genere fornite da sorgenti diverse).

Quindi, nel caso in cui l'utente abbia deciso di effettuare una query "concept" per una o più feature, il numero, il tipo e l'ordine dei join inseriti relativamente alle sole tabelle di id_translation e di feature cambia lievemente.
Perché si possa effettuare una query concept è necessario che l'utente abbia selezionato almeno un attributo dalla tabella di feature e/o di id_translation della feature su cui vuole effettuare la query concept.

Si distinguono quindi tre scenari:

- l'utente ha selezionato attributi sia dalla tabella di feature che dalla tabella di id_translation
- l'utente non ha selezionato attributi dalla tabella di id_translation.
- l'utente non ha selezionato attributi dalla tabella di feature.

Caso 1: tabella di feature e tabella di id_translation

In questo caso, nella lista delle tabelle inserite nella query sono presenti sia la tabella di feature che la tabella di id_translation.
Per effettuare correttamente una query "*concept*", prima di svolgere tutti i join come avviene normalmente, è necessario estrarre dalla tabella di feature o dalla tabella di id_translation le tuple che hanno lo stesso *concept_oid*.
In questo caso, quindi, essendo presenti entrambe le tabelle, prima di effettuare tutti i join rimanenti, è necessario inserire il join tra la tabella di *feature* e quella di *id_translation* ponendo come condizione l'uguaglianza tra i due *concept_oid*.
L'algoritmo, quindi, quando incontra una tabella di *id_translation*, anziché inserire il normale join tra gli *"_oid*", nel caso di una query *concept* inserisce il join utilizzando i "*concept_oid".*

Un esempio di concept query in questo caso (viene riportata solo la parte della clausola FROM) è mostrato in Tabella 7.

```
FROM public.biological_function_feature
RIGHT JOIN public.biological_function_feature_id_translation ON
public.biological_function_feature.biological_function_feature_c
oncept_oid =
public.biological_function_feature_id_translation.translated_bio
logical_function_feature_concept_oid
LEFT JOIN public.go_biological_process ON
public.biological_function_feature.biological_function_feature_o
id=public.go_biological_process.go_biological_process_oid
LEFT JOIN public.biological_function_feature_alternative_id ON
public.biological_function_feature.biological_function_feature_o
id=public.biological_function_feature_alternative_id.biological_
function_feature_oid
```

Tabella 7 - Clausola FROM di query concept con tabella di feature e di id_translation

Caso 2: nessun campo di tabella di id_translation selezionato

Nel caso in cui non siano stati selezionati campi della tabella di id_translation (quindi nella lista delle tabelle sia presente soltanto la tabella di feature, e *non* quella di id_translation corrispondente), per individuare le tuple con lo stesso "concept_oid" si effettua un join "circolare" sulla tabella di feature stessa, selezionando le tuple con lo stesso *concept_oid*. Anche questo join viene effettuato prima di tutti gli altri join della query.

Un esempio di concept query in questo caso (viene riportata solo la parte della clausola FROM) è mostrato in Tabella 8.

```
FROM public.biological_function_feature
LEFT JOIN public.biological_function_feature AS
biological_function_feature_ConceptAlias ON
public.biological_function_feature.biological_function_feature_c
oncept_oid =
biological_function_feature_ConceptAlias.biological_function_fea
ture_concept_oid
LEFT JOIN public.go_biological_process ON
public.biological_function_feature.biological_function_feature_o
id=public.go_biological_process.go_biological_process_oid
LEFT JOIN public.biological_function_feature_alternative_id ON
public.biological_function_feature.biological_function_feature_o
id=public.biological_function_feature_alternative_id.biological_
function_feature_oid
```

Tabella 8 - Clausola FROM di query concept con soltanto tabella di feature

Caso 3: nessun campo di tabella di feature selezionato

In questo caso ci si riconduce al caso 2, effettuando un join "circolare" sulla tabella di id_translation stessa, selezionando le tuple con lo stesso *concept_oid*.

Anche questo join viene effettuato prima di tutti gli altri join della query.

Nel caso di doppia (o multi) feature, il procedimento è lo stesso, e viene svolto per tutte le feature per cui l'utente ha espresso la volontà di effettuare una query ***concept***.

6.1.5 Query di conteggio dei risultati

Dopo che la query principale è stata generata ed eseguita, il sistema genera la query di conteggio dei risultati.

In una prima fase di visualizzazione, viene fornito all'utente un valore approssimato del conteggio dei risultati, ottenuto eseguendo un parsing di una query di *explain* della query appena eseguita.

Dopodichè viene generato il testo della query di *count* vera e propria. Questa query non viene eseguita subito, ma viene eseguita in background con una chiama *AJAX* dopo che all'utente sono già stati mostrati i risultati.

Il testo della query di count è riportato in Tabella 9.

```
SELECT COUNT(*) FROM (<query executed>) AS result
```

Tabella 9 - Testo della query di conteggio dei risultati

Si è visto che, con questa formulazione della query di conteggio, il sistema conta anche le righe dei risultati con tutti i valori a *null* (cosa che non avveniva con altre formulazioni della query). Il numero di righe con tutti i valori a *null* è considerato di interesse per l'utente e di conseguenza è stato contato.

Nel caso in cui l'utente, in fase di composizione, abbia selezionato la modalità di "*Estimated count*", il sistema esegue la query di count standard per fornire all'utente il numero esatto di risultati con una soglia di durata (indicata nel file di configurazione *settings.xml*). Se il tempo impiegato dalla query supera questa soglia, allora la query viene interrotta e all'utente rimane indicato il valore approssimato, con un avviso di warning. Se invece questa query giunge a un risultato in un tempo inferiore a quello indicato dalla soglia, il valore di count approssimato viene sostituito con il valore esatto, e questo viene indicato all'utente con un avviso.

Nel caso in cui l'utente, in fase di composizione, abbia invece selezionato la modalità di "*Exact count*" la procedura è la stessa, senza la soglia di durata. Si aspetta, quindi, in background, il termine della query di count. Quando questa giunge a un risultato, come in precedenza il valore di count approssimato viene sostituito con il valore esatto, indicandolo all'utente con un avviso.

Al termine di questa procedura, i dati relativi al numero esatto di tuple ottenute vengono passati tramite *JSON* al plugin di visualizzazione.

6.2 Generazione interfaccia di selezione degli attributi

In questo paragrafo viene descritta la modalità di generazione della pagina di selezione degli attributi, utilizzata, in modo diverso, sia nella modalità visuale che in quella grafica di generazione delle query. In Figura 22 è riportato uno screenshot di una pagina di questa modalità.

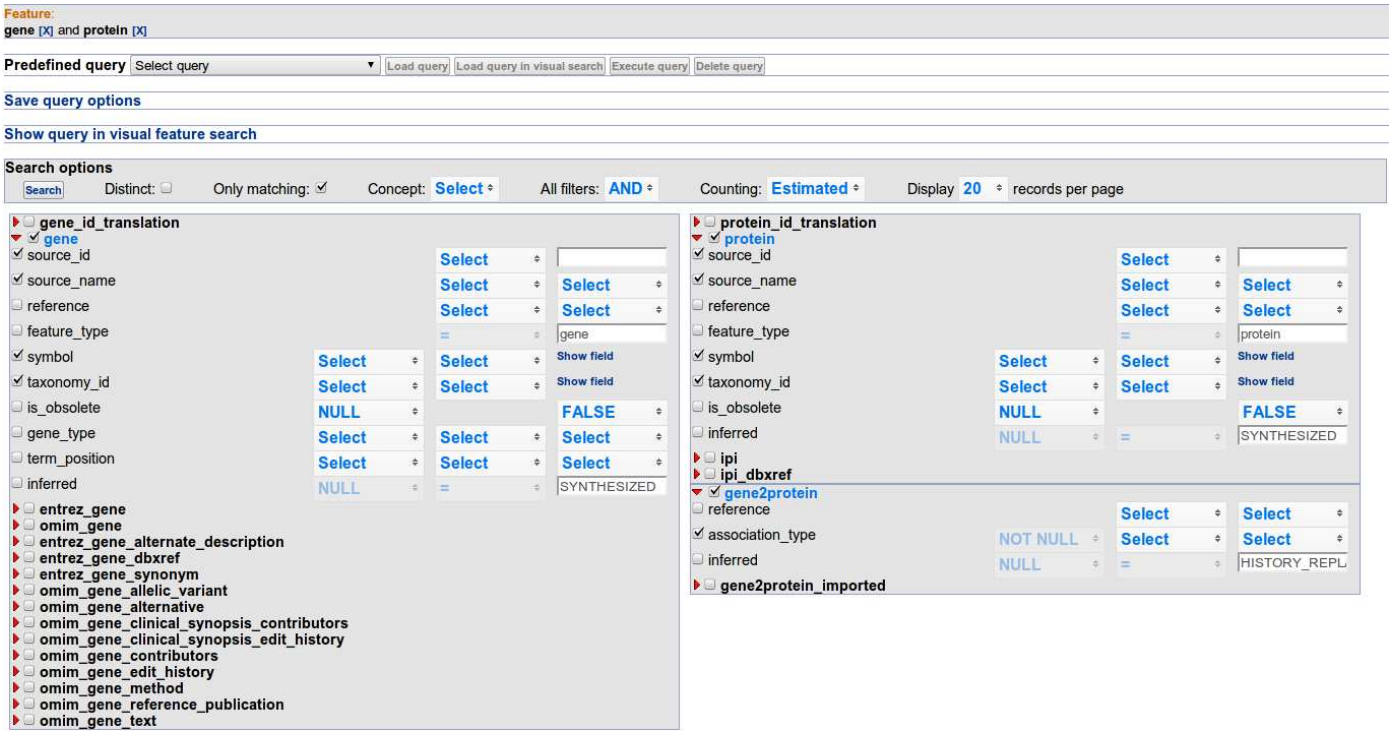


Figura 22 – Screenshot di una pagina della modalità visuale di selezione degli attributi

Quando l'utente seleziona una feature dalla pagina generata dalla servlet *SelectFeature*, viene eseguita la servlet *GetFeatureAttributes*.

Questa servlet si occupa di generare la lista dei nomi delle tabelle da mostrare relative alla singola (o alla coppia) di features e di passarla alla pagina *selectAttributes.jsp*.

In cima alla pagina vengono mostrate le feature selezionate. Nel caso di multipla feature, è possibile eliminare una delle feature selezionate, cliccando sulla *[X]* al suo fianco (in Figura 23 ne è riportato un esempio).

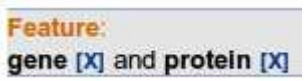


Figura 23 - Rimozione di una feature dalla query

Nel caso di singola feature, viene mostrato all'utente un menu di selezione di un'eventuale feature associata, per espandere la query da singola a multipla feature.
Vengono poi mostrare le "*Search options*" (Figura 24), cioè alcune opzioni per effettuare la query:

- ***Distinct:*** se selezionato, la query verrà effettuata con la clausola "distinct"
- ***Only matching:*** se selezionato, i join della query saranno tutti di tipo *INNER JOIN*. *OUTER*, altrimenti.
- ***Concept:*** in questo menu è possibile selezionare se effettuare o meno una query concept (per dettagli vedi sezione 6.1.4) e, nel caso di multipla feature, per quale/i feature effettuare la query concept stessa.
- ***All filters***: definisce quale connettore usare tra le condizioni in where (AND oppure OR)
- ***Counting***: definisce quale modalità di conteggio dei risultati utilizzare. Per i dettagli, vedi sezione 6.1.5.
- ***Display***: definisce quanti risultati mostrare per pagina nella visualizzazione dei risultati

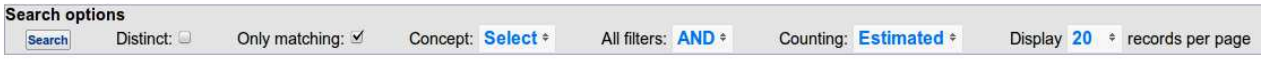


Figura 24 - Menu delle search options

Di seguito alle *search options* viene mostrata la lista delle tabelle di cui si possono selezionare gli attributi, che sono:

- nel caso di singola feature:
  - la tabella di *id_translation*
  - la tabella di *feature*
  - le tabelle addizionali di feature
  - le tabelle di *sorgente*
  - le tabelle addizionali di sorgente

- nel caso di doppia feature:
  - per ogni feature, come nel caso singola feature
  - la tabella di associazione tra le due feature, con le sue tabelle addizionali, importate e addizionali importate

La lista delle tabelle viene visualizzata con una chiamata, per ogni tabella, alla funzione *FeatureAttributesTemplate.getTableNameRow(),* che si occupa di generare il codice html relativo alla riga del form di quella tabella.

Le tabelle di feature e di associazione vengono mostrate con tutti gli attributi, mentre delle altre tabelle viene mostrato solo il nome. Cliccando sulla freccia a sinistra del nome di ogni tabella, vengono mostrati tutti gli attributi selezionabili di quella tabella.

Gli attributi selezionati di default per le tabelle mostrate sono definiti nel file default_fields.xml (vedi sezione 5.4).

Questi attributi non vengono caricati in fase di generazione della pagina, ma sono caricati tramite una chiamata *AJAX* alla servlet *GetTableFields* quando si clicca sulla freccia di apertura della tabella. Questa separazione delle fasi di caricamento degli attributi di ogni tabella si è resa necessaria per diminuire i tempi di caricamento iniziali della pagina di selezione degli attributi.

La servlet *GetTableFields*, dopo aver raccolto i dati degli attributi dal database, chiama a sua volta la funzione *FeatureAttributesTemplate.getTableAttributeRow()*, che si occupa di popolare e mettere in output il form html di ogni attributo.

E' all'interno di questa funzione che vengono presi dal database i valori dei campi codificati e messi all'interno della cache del server, con il nome *<nome_attributo>_encoded*. Il valore di questi attributi è una *HashMap* le cui chiavi sono i valori codificati dei campi, mentre i valori della mappa sono i valori decodificati associati al valore codificato (vedi sezione 5.1.2).

In cima alla pagina, infine, vengono mostrati i menù di gestione (salvataggio e caricamento) delle query e la possibilità di mostrare la query in composizione nella modalità visuale (Figura 25).

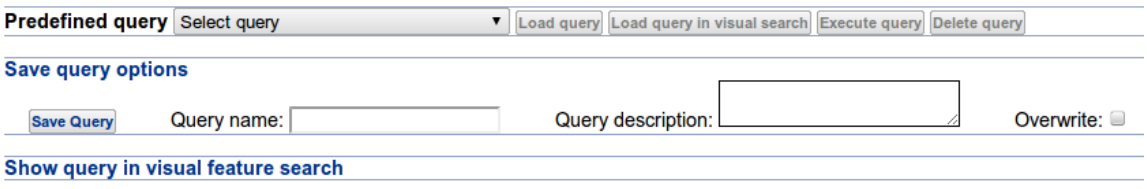


Figura 25 - Menu di gestione e estensione query

6.2.1 Criterio di selezione degli attributi

Per la composizione della pagina di selezione degli attributi sono stati seguiti alcuni criteri e alcuni algoritmi per ottimizzarne il caricamento e per evitare ridondanze inutili nel mostrare gli attributi stessi.

Gli attributi delle tabelle di *feature* e di *associazione* vengono mostrati tutti, sempre.
Per le altre tabelle, invece, viene controllato che i nomi di tutti i loro attributi non siano già contenuti nelle loro tabelle "padre"(vedi sezione 6.1.1). Se c'è almeno un attributo che non è contenuto nelle tabelle "padre", allora la tabella, con tutti i suoi attributi, viene mostrata nell'elenco. Altrimenti la tabella non viene considerata.

Per mantenere la retro compatibilità con istanze precedenti del GPDW, è stato necessario inserire poi alcune regole di inclusione delle tabelle nella lista: per quando riguarda le tabelle di associazione importate (o addizionali di associazione importate), se queste non possiedono l'attributo "*association_oid*", allora non verranno visualizzate nella lista delle tabelle selezionabili. Questo perché l'attributo "*association_oid*" è stato aggiunto in versioni recenti del database, ed è necessario per poter generare ed eseguire correttamente le query che coinvolgono tabelle di associazione importate oppure tabelle addizionali di associazione importate.

6.2.2 Modalità di conteggio dei risultati

Per definire la modalità di conteggio selezionata di default, vengono seguiti alcuni criteri:

- Se tra le feature selezionate non è presente nessuna entità biomolecolare, allora la modalità selezionata sarà di "***Exact count***", cioè verrà effettuata una query di count standard.
- Se invece una delle feature (o entrambe) è una entità biomolecolare, la modalità selezionata sarà di "***Estimated count***" (cioè effettuando una query di explain e prendendo dal suo risultato il numero di tuple stimato)

Questa distinzione si è resa necessaria in quanto una query di count "standard" si è rivelata essere molto lenta su tabelle con molte tuple e su query con molti filtri. Da qui l'esigenza di resituire all'utente, almeno all'inizio, un valore approssimato in tempi ragionevolmente brevi.

Nel caso in cui venga selezionata la modalità di "*Estimated count*", il sistema, dopo aver eseguito la query e indicato all'utente il valore di tuple approssimato, cercherà comunque di eseguire la query di count standard per fornire all'utente il numero esatto di risultati. Se il tempo impiegato da questa query, però, supera una certa soglia (definita in un file di configurazione xml, vedi sezione 5.4), allora la query viene interrotta e all'utente rimane indicato il numero approssimato, con un avviso di warning.

Nel caso invece di "*Exact count*" la procedura è la stessa, senza la soglia di durata. Si aspetta, quindi, in background, il termine della query di count "standard". Quando questa giunge a un risultato, il valore di count approssimato viene sostituito con il valore esatto, e questo viene indicato all'utente con un avviso.

6.3 Modalità grafica di composizione delle query

La modalità chiamata "grafica" di composizione della query è stata pensata per permettere agli utenti di comporre delle query che comprendessero più di due features.

E' infatti immediato immaginare che una pagina come quella descritta nella sezione precedente, dovendo mostrare le tabelle e gli attributi relativi a più di due feature, diventerebbe piena di informazioni e difficilmente comprensibile.

E' stato quindi pensato di generare un grafo delle features, sia per permettere agli utenti di avere un visione di "insieme" dei dati presenti nel GPDW (e le loro associazioni), sia per permettere una composizione di query multifeature più agevole.

A oggi, il grafo delle features è quello riportato in Figura 26. In rosso sono riportate le entità biomolecolare, in arancione le feature biomediche (vedi sezione 2.1).

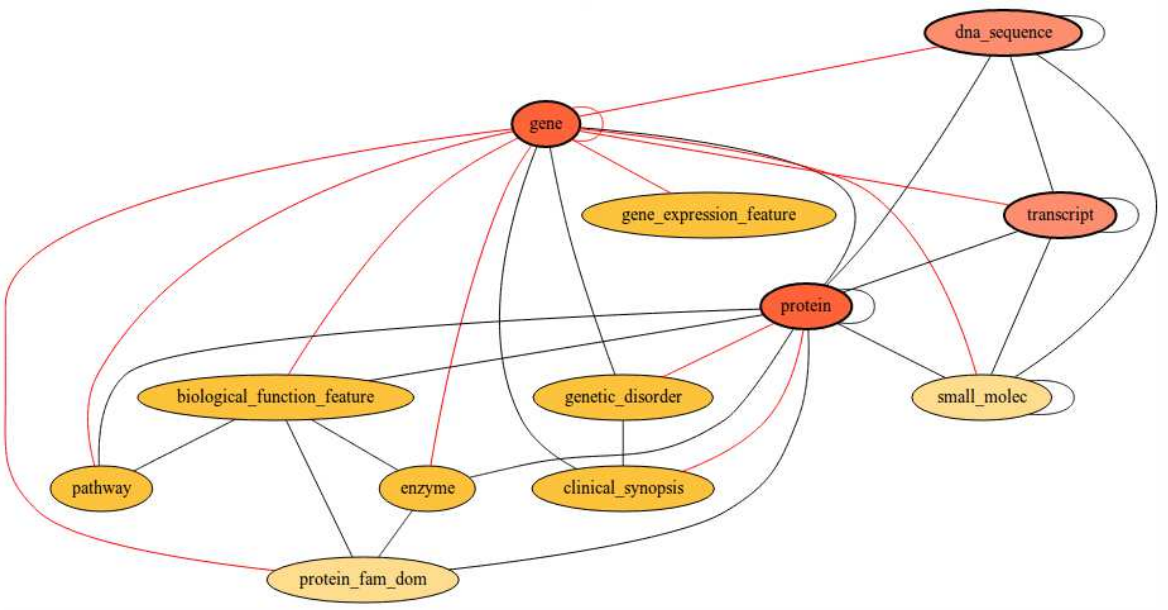


Figura 26 - Grafo delle features

L'utente, cliccando su ciascun nodo di questo grafo, può selezionare una feature da includere nella sua query.

Quando l'utente seleziona una feature, tutti i nodi a lei non collegata vengono "spenti", e rimangono selezionabili solo i nodi (e gli archi) associati alla feature selezionata. In Figura 27 viene riportato lo stato del grafo dopo la selezione della feature "*biological_function_feature*".

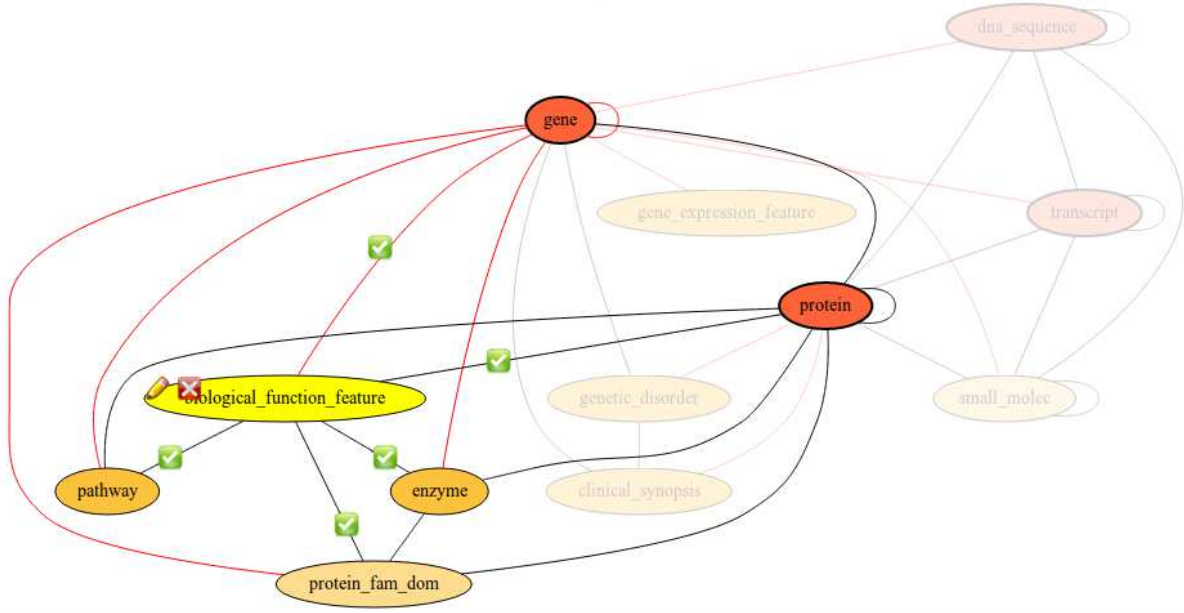


Figura 27 – Stato del grafo dopo la selezione di una feature ("biological_function_feature")

Cliccando sull'icona di spunta  , è possibile selezionare un arco del grafo. Selezionando un arco, vengono attivate (e quindi selezionate) entrambe le features da lui collegate (Figura 28).

Analogamente, attivando un nodo ancora selezionabile, vengono selezionati tutti gli archi tra questo nodo e tutte le features attive.

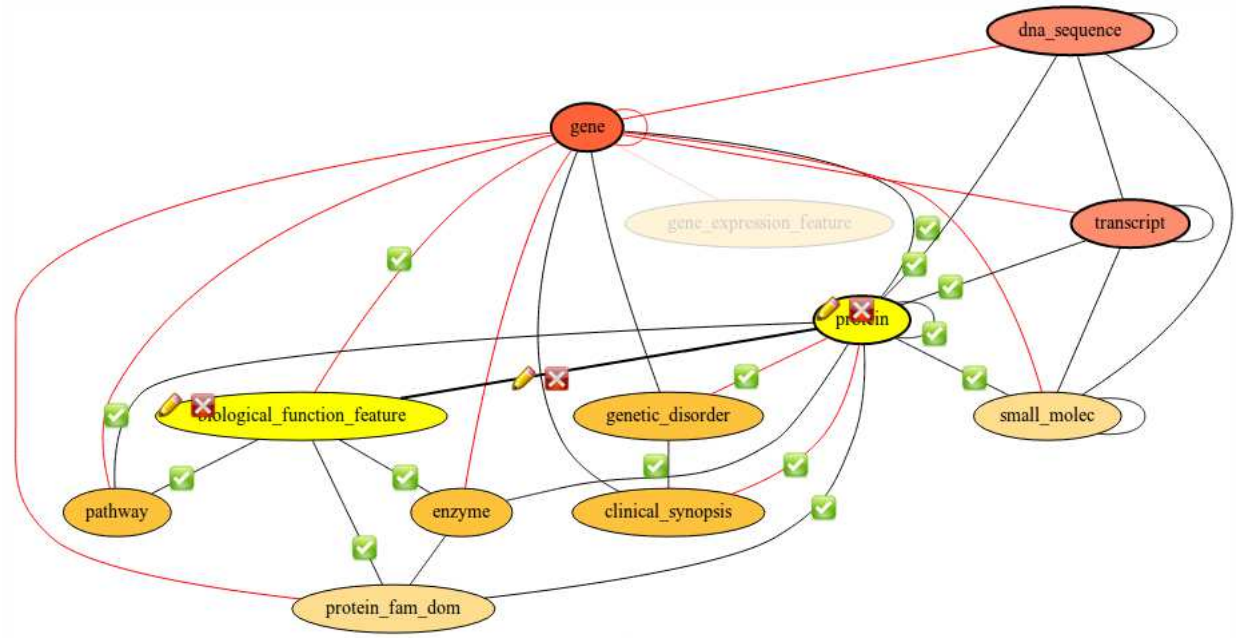


Figura 28 - Stato del grafo dopo la selezione di una seconda feature ("protein")

Cliccando, per ogni nodo o arco selezionato, sull'icona  , il nodo (o l'arco) corrispondente viene disattivato e rimosso dalla query.

Cliccando invece sull'icona  , viene mostrato all'utente il dialog di selezione degli attributi (Figura 29) del corrispondente nodo o arco, che viene generato in modo analogo a quanto descritto nella sezione 6.2.

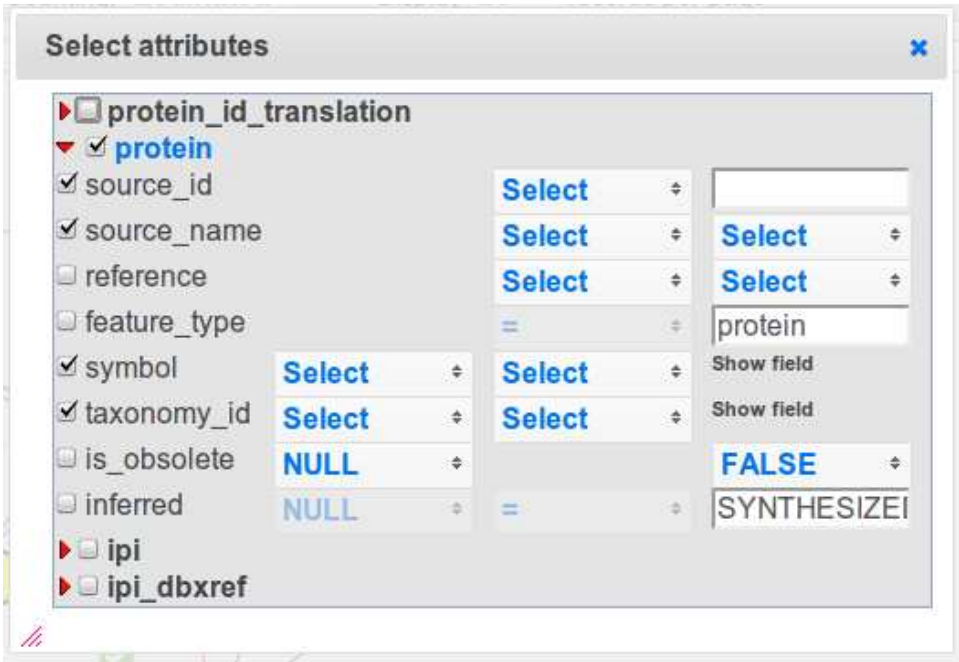


Figura 29 - Dialog di selezione degli attributi

Tramite questo dialog è possibile selezionare e filtrare gli attributi che si vogliono inserire nella query.

Il resto della pagina è analogo a quanto visto nella sezione 6.2 per la modalità "visuale" di composizione della query: è possibile salvare e caricare le query, selezionare le "Search options".

6.3.1 Generazione del grafo delle features

Le generazione del grafo delle features avviene in due fasi, eseguite da due servlet distinte: *GraphGenerator* (che si occupa della generazione del file svg del grafo) e *GraphParser* (che analizza il file svg generato vi aggiunge informazioni necessarie per la gestione dell'interazione con l'utente), appartenenti entrambi al package *metadataGraph*.

Generazione del grafo

La servlet *GraphGenerator* preleva i dati dello schema *metadata* del GPDW, in particolare dalle tabelle *source2feature_association* e *feature_association*, e li elabora associando un nodo del grafo a ogni feature e un arco a ogni loro associazione.

Quindi, attraverso la classe *Graph* del package *graph*, genera il file .dot che viene passato, eseguendo un comando tramite il server, a un'istanza del programma di gestione dei grafi ***Dot***, il quale crea un file *svg (graph.svg)* corrispondente al grafo indicato.
Il layout del grafo può essere modificato tramite l'enumerazione Graph.NodeType, definendo l'aspetto dei nodi, a seconda del loro tipo.
I tipi di nodo definiti sono i seguenti:

- **feature biomedica** (BIOMEDICAL_FEATURE)
- **feature biomedica sintetizzata** (BIOMEDICAL_FEATURE_SYNTHESIZED)
- **entità biomolecolare** (BIOMOLECULAR_ENTITY)
- **entita biomolecolare sintetizzata** (BIOMOLECULAR_ENTITY_SYNTHESIZED)

Dopo che il grafo è stato generato, viene passato alla servlet GraphParser per essere ulteriormente elaborato.

Parsing del grafo

La servlet GraphParser legge il file graph.svg e inserisce dei dati in formato xml a ogni nodo e a ogni arco. Questi dati saranno necessari per l'interazione con il grado da parte dell'utente.
Nel file svg, ogni nodo o arco è identicato da un gruppo svg (tag <g>). Questi dati aggiuntivi vengono inseriti al termine di ogni tag <g>. Ogni gruppo è identificato da un *id*, che nel codice è chiamato *javascript id*.
Per i nodi i dati aggiunti sono mostrati in Tabella 10.

```
<nodeDbId> The id of the feature into the database </nodeDbId>
<featureTableName>
The feature table name taken from metadata schema
</featureTableName>
<nodeStyle> The graphical style of the node into the graph </nodeStyle>
<href> Value of href attribute (default: "javascript:") </href>
<onclick> Value of onclick attribute </onclick>
```

Tabella 10 - Dati xml aggiuntivi per ogni nodo

Per gli archi invece i dati aggiunti sono mostrati nella Tabella 11.

```
<nodeStart> Database id of the start node. </nodeStart>
<nodeEnd> Database id of the end node. </nodeEnd>
<edgeStyle> Graphical style. </edgeStyle>
<href></href>
<onclick> </onclick>
```

Tabella 11 - Dati xml aggiuntivi per ogni arco

Di seguito, una spiegazione dei dati indicati:

- ***nodeDbId***: per "*database id*" (*dbId*) si intende, per ogni feature, il valore del campo "*feature_id*" nella tabella *metadata.feature*. Normalmente è un numero intero. Nel codice, è usato come riferimento alla feature stessa.
- ***featureTableName***: è il valore del campo *feature_table_name* della tabella *metadata.feature*, e indica il nome testuale della feature
- ***nodeStyle***: è lo stile del nodo quando questo è attivabile (non attivo e non nascoso)
- ***href*** e ***onclick***: contengono i valori di questi attributi nello stato iniziale del grafo
- ***nodeStart*** e ***nodeEnd***: contengono il database id dei nodi collegati dall'arco in questione

Quando questi dati sono stati inseriti nel grafo, la servlet genera un nuovo file, *graphFinal.svg*, che poi sarà il file definitivo che verrà esposto per l'interazione con gli utenti nella pagina *visualSearch.jsp*.

6.3.2 Interazione con il grafo

Dopo che il grafo è stato generato, questo deve interagire con l'utente.
Questa interazione è stata realizzata tramite diversi script in linguaggio JavaScript, che prelevano e impostano alcuni attributi del grafo svg.
Questa applicazione javascript può essere divisa in due livelli: il livello "*dati*" e il livello "*grafico*".

Livello dati

I dati sono strutturati in diverse classi javascript:

- **Node**: questa classe identifica un nodo all'interno del grafo. I suoi attributi sono "*database id*", "*node type*" (analogamente a quanto visto nella sezione precedente) e una *lista* contenente tutte le feature "*attive*" (cioè già selezionate dall'utente) collegate a questo nodo.

- **Edge**: questa classe identifica un arco all'interno del grafo. Contiene i "*database id*" dei nodi che collega.
- **NodeList** e **EdgeList**: sono liste degli oggetti appena descritti

I dati sono salvati in due liste:

- **activeFeatures**: è una NodeList contenente tutte le features selezionate dall'utente
- **activableFeatures**: è una NodeList contenente tutte le feature collegate a una feature attiva (e quindi selezionabili dall'utente).
- **activeEdges**: è una EdgeList contenente tutti gli archi selezionati dall'utente

Livello grafico

La gestione grafica del grafo è realizzata tramite tre array:

- **nodesToActivate**: è un array di javascript id (vedi sezione precedente) contenente gli id di tutti i nodi che devono essere attivati alla successiva chiamata della funzione repaint() del grafo, la quale si occupa di ridisegnare il grafo secondo i cambiamenti dei dati
- **nodesToReset**: è un array contenente gli id di tutti i nodi che devono essere resettati (cioè riportati alla loro condizione iniziale) alla successiva chiamata della funzione repaint() del grafo
- **nodesToHide**: è un array contenenente gli id di tutti i nodi che dono essere nascosti (cioè resi trasparenti e inattivi) alla successiva chiamata della funzione repaint() del grafo

Interazione con l'utente

Quando il grafo viene caricato per la prima volta, viene mostrato con tutte le features attivabili, e tutti gli archi inattivi.

Quando l'utente clicca su un nodo, questo nodo viene attivato e evidenziato graficamente (diventa "giallo"). Ora l'interazione può proseguire in due modi:

- selezionando un altro *nodo* vengono effettuate queste operazioni:
  - la feature attivata viene aggiunta alla lista *activeFeatures* a rimossa dalla lista *activableFeatures*, e il nodo in questione viene aggiunto alla lista *nodesToActivate*
  - tutte le features collegate a questa features vengono rese attivabili (cioè aggiunte alla lista *activableFeatures*)

  - viene cambiato l'attributo *onClick* di questo nodo, tramite il quale ora verrà avviata la procedura di deselezione della feature
  - viene effettuata una chiamata ajax alla servlet *GetVisualSearchTableFields*, che si occupa di prelevare dal database tutti gli attributi realtivi alle tabelle di questa feature; viene generato il dialog corrispondente e messo in background
  - viene effettuata una chiamata alla funzione *repaint()*
- selezionando un *arco* vengono effettuate le seguenti operazioni:
  - viene attivato l'arco selezionato e viene aggiunto alla lista *activeEdges*
  - le feature collegate da questo arco vengono attivate entrambe
  - viene cambiato l'attributo *onClick* di questo arco, tramite il quale ora verrà avviata la procedura di deselezione dell'arco
  - viene effettuata una chiamata ajax alla servlet *GetVisualSearchTableFields*, che si occupa di prelevare dal database tutti gli attributi realtivi alle tabelle relative all'associazione indicata dall'arco; viene generato il dialog corrispondente e messo in background
  - viene effettuata una chiamata alla funzione *repaint()*

6.4 Esecuzione query e visualizzazione dei risultati

Quando ha completato la generazione della query, l'applicazione (tramite la servlet *ComposeQuery*) esegue la pagina *showAddTable.jsp*, che ci occupa dell'esecuzione della query e della visualizzazione dei risultati.

6.4.1 Esecuzione query

Per quanto riguarda l'esecuzione della query, la pagina *ShowAddTable.jsp* esegue (tramite il plugin JQuery *DataTable*, vedi sezione 4.5.3) una chiamata *AJAX* alla servlet *DataTableGPKB*.
Questa servlet, oltre ad inizializzare alcuni parametri necessari alla visualizzazione, genera il testo della query completa unendo le parti generate dalla servlet *ComposeQuery* e aggiungendo le clausole ***ORDER BY*** (di default i dati vengono ordinati secondo il primo attributo selezionato) e ***LIMIT*** (secondo il valore indicato dall'utente in fase di composizione) e la esegue.
Lo Statement con cui viene eseguita la query viene salvato nella sessione dell'utente, per poter permettere la sua cancellazione nel caso in cui venga lasciata la pagina di visualizzazione prima del termine della query stessa.

6.4.2 Analisi e preparazione dei risultati per la visualizzazione

Dopo che la query è stata eseguita, i risultati ottenuti vengono analizzati e modificati per essere visualizzati.

Campi codificati

Ogni valore di un campo codificato viene sostituito con il suo valore non codificato. Questa associazione *(campo codificato, valore non codificato)* è presente nella sessione dell'utente in quanto è stata salvata durante le generazione della pagina di composizione query (vedi sezione 5.1.1). Se, per qualsiasi motivo, questa associazione non è presente, il valore non codificato viene preso con una query allo schema flag del database (*DBFlags.getFieldRealValue()*).
Se il campo codificato è di tipo bitmap e contiene più di un bit a 1 (es: *reference_file*), significa che nella tabella di *flag* avrò più valori corrispondenti a questo campo, cioè i valori

corrispondenti a tutti i valori bitmap del campo con uno solo dei bit a 1 attivo e tutti gli altri portati a 0.
I loro valori non codificati verranno mostrati tutti, separati da una virgola.

Campi a cui sono associati collegamenti (hyperlinks)

A ogni valore di un campo contente un collegamento viene associato il collegamento stesso in modo che, cliccando sul valore mostrato, venga mostrata, in un'altra tab del browser, la pagina collegata.

Per ogni valore di un campo che termina con *"_id"*, inoltre, nello schema dei metadati del database è potenzialmente presente un collegamento ipertestuale che rimanda a una pagina esterna al GPKB che fornisce informazioni aggiuntive sul valore di questo campo.
Al momento attuale la ricerca nei metadati di questo collegamento è implementata per i campi *source_id* e *<feature_name>_id* e per tutti i campi che terminano con *"_id"* delle tabelle di associazione (importate e non).
I collegamenti ipertestuali sono salvati nelle tabelle *metadata.source2feature_display_url* e *metadata.feature_association_display_url*. Nella prima tabella sono identificati da una coppia *(source_id, feature_id)* e il campo *display_class,* mentre nella seconda tabella sono identificati dalla coppia *(feature1_id, feature2_id)* e il campo *display_class*. Per quanto riguarda i campi delle tabelle di associazione importate, è necessario un ulteriore campo, il campo *reference*.

Il sistema, quindi, per ogni campo *source_id* e *<feature_name>_id* (o **_id* nel caso di tabelle di associazione) selezionato dall'utente, associa a ognuno un insieme di *feature_id* (corrispondenti alla feature di quel campo), con delle query ai metadati (*DBGetters.getSourceIdFeatureId()*). Queste query prelevano dalla tabella *metadata.feature* tutti i *feature_id* corrispondenti alla data tabella di feature. Per come è strutturato il GPDW (vedi sezione 2.1), una feature può avere delle "sottofeature", ognuna con il suo relativo id. Di conseguenza, in questo momento dell'algoritmo, a ogni attributo vengono associati tutti i feature_id associati alla tabella a cui appartiene, quello della feature principale e quello delle sue "sottofeature".
I *feature_id* possono essere più di uno anche nel caso di campi di tabelle di associazione, dove i feature_id a loro attribuiti saranno gli id delle feature associate dalla tabella di associazione.

Questo avviene prima di considerare ogni riga dei risultati ottenuti con la query composta dall'utente (non è necessario conoscere i valori dei campi richiesti ma soltanto i nomi degli stessi). I flowchart di questo algoritmo sono riportati in Figura 30 e Figura 31.

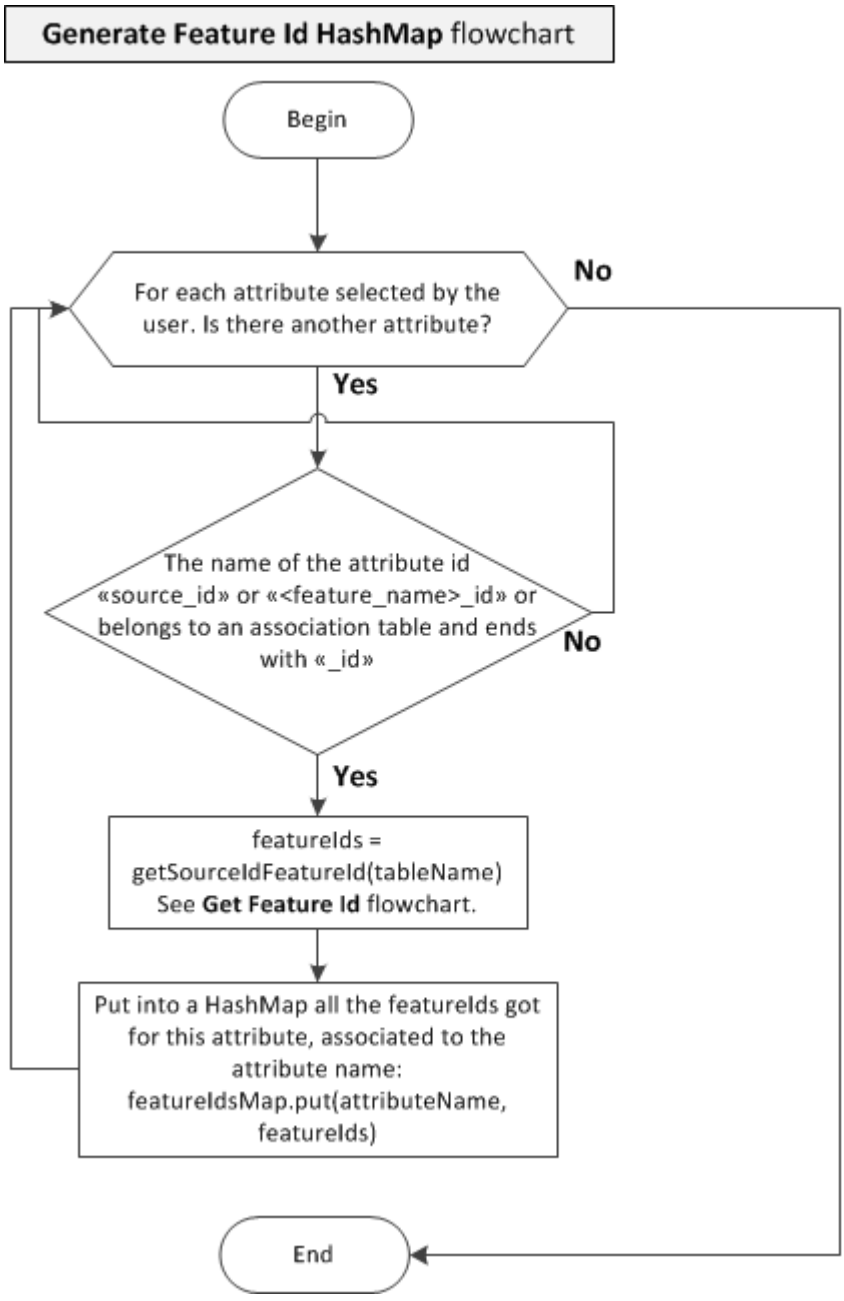


Figura 30 - Flow chart dell'algoritmo di generazione della HashMap che associa a ogni attributo selezionato dall'utente i suoi feature id corrispondenti

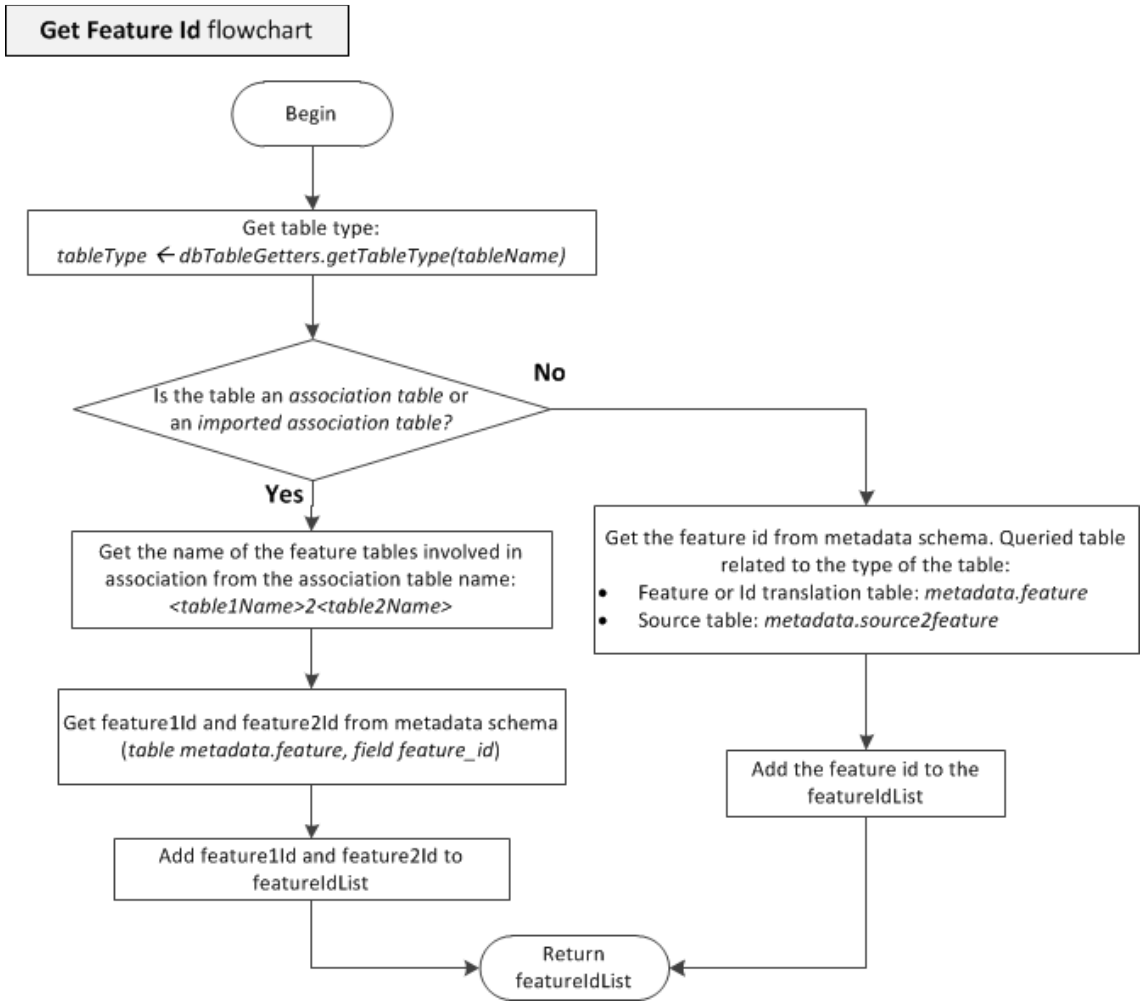


Figura 31 - Flow chart della funzione getSourceIdFeatureId che preleva dal dabatase i feature id relativi a una data tabella del GPDW

Dopodichè, per ogni riga dei risultati, per ogni campo *source_id* e *<feature_name>_id* (o **_id*) viene individuato il relativo valore del campo *source_name* oppure *<feature_name>_source.* Questi campi contengono il valore del campo *source_id* della coppia *(feature_id, source_id)* delle tabelle dei metadati necessaria per ottenere l'indirizzo del collegamento ipertestuale (*metadata.source2feature_display_url*). Inoltre, se è presente il campo *feature_type* nella riga dei risultati, da qui viene preso il feature_id corretto nell'insieme dei feature_id precedentemente ottenuti (di feature e sottofeature) per questo campo. Se non è presente il campo feature_type, significa che la feature in esame non posside sottofeature e di conseguenza a questo attributo è già stato associato un unico *feature_id*.

Nel caso di tabelle di associazione, sono presenti due coppie *(<feature_name>_id, <feature_name>_source)*. I *<feature_name>_id* sono ottenuti con il metodo precedentemente

esposto. I valori dei campi *<feature_name>_source,* invece, sono salvati in una struttura dati e vengono popolati cercandoli nella riga dei risultati esaminata.

Se si sta cercando l'URL di un attributo di una tabella di associazione (importata e non), viene individuato anche il valore del campo *reference*.

Ottenuta la (e/o le) coppie *(source_id, feature_id)*, queste vengono passate alla funzione *DBGetters.getSourceIdURL()*, che si occupa di ottenere il corretto URL dalle tabelle *metadata.source2feature_display* oppure *metadata.source2feature_association_display*.

Se alla funzione *DBGetters.getSourceIdURL()* viene passata una sola coppia *(source_id, feature_id)*, allora l'URL corrispondente verrà ricercato soltanto nella tabella *metadata.source2feature_display_url*.

Verranno ricercate le tuple corrispondenti alla coppia *(source_id, feature_id)* il cui attributo *"display_class"* ha valore *"main"*.

Se invece alla funzione *DBGetters.getSourceIdURL()* vengono passate due coppie *(source_id, feature_id)*, significa che si sta ricercando un URL di un attributo di una tabella di associazione. Quindi, prima viene ricercato per ogni coppia un URL nella tabella *metadata.source2feature_display_url* (come in precedenza). Se questo URL non viene trovato, allora viene cercato un URL corrispondente alla coppia *(feature1_id, feature2_id),* i cui valori sono ottenuti unendo i feature_id delle due coppie passate in precedenza, nella tabella *metadata.feature_association_display_url*, il cui attributo *"display_class"* ha come valore il nome dell'attributo di cui si ricerca il collegamento ipertestuale.

Se, in entrambi i casi, viene trovato un URL, viene aggiunto un collegamento ipertestuale al valore del campo *source_id* o *<feature_name>_id*. Se non viene trovato nessun URL, viene mostrato il valore del campo senza nessun collegamento.

In Figura 32 e Figura 33 sono riportati i flow chart di questo algoritmo.

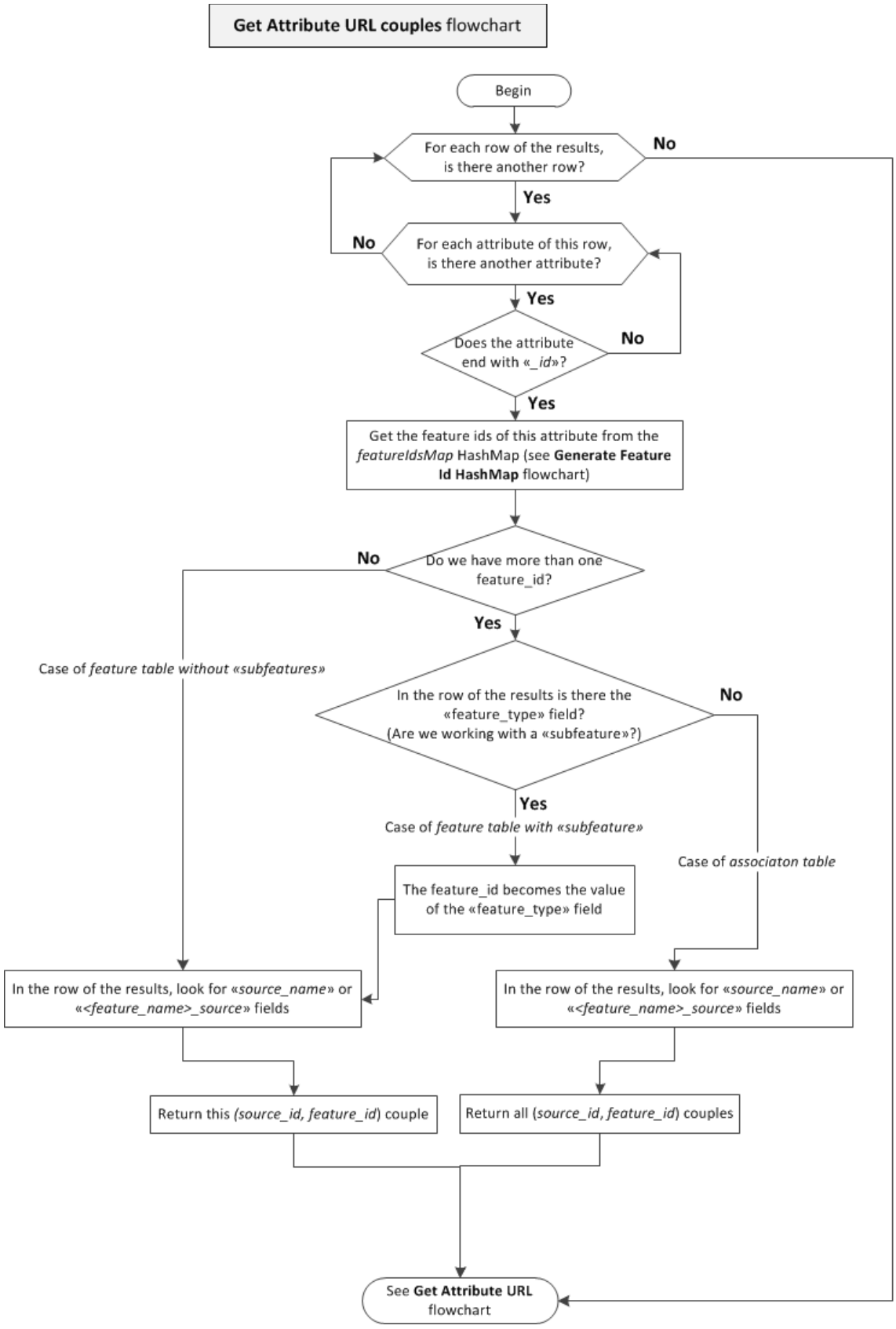


Figura 32 - Prima parte dell'algoritmo di ottenimento dal database dell'URL da associare a un campo il cui nome termina con il suffisso "_id": ricerca delle coppie (source_id, feature_id)

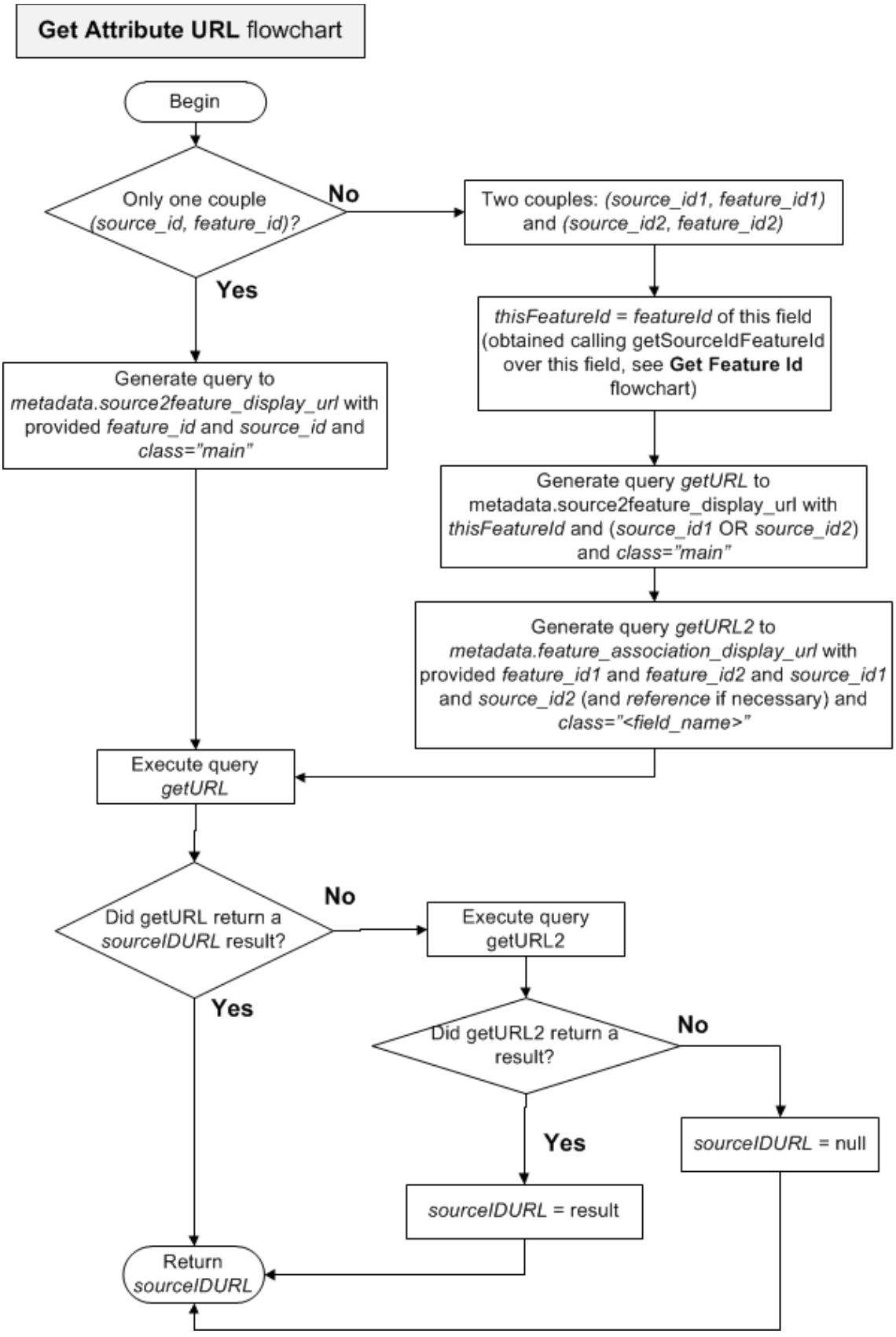


Figura 33 – Seconda parte dell'algoritmo di ottenimento dal database dell'URL da associare a un campo il cui nome termina con il suffisso "_id": generazione delle query di interrogazione ai metadati

6.4.3 Visualizzazione dei risultati

Per quanto riguarda la visualizzazione dei risultati, è stato utilizzato il plugin *JQuery DataTables*, che permette una dinamica visualizzazione tabellare dei dati.

Ogni volta che viene richiesto un nuovo ordinamento dei dati, oppure si passa a una nuova pagina della tabella, viene eseguita nuovamente la servlet *DataTableGPKB* che si occupa di mandare in esecuzione di nuovo la query principale con i diversi valori di ***ORDER BY*** e ***LIMIT***. La query di count, invece, viene eseguita una volta sola, alla prima esecuzione della servlet.

Filtro dei risultati

Dalla pagina di visualizzazione dei risultati è possibile filtrare ulteriormente ciò che si è ottenuto impostando nuovi valori dei filtri già selezionati oppure selezionandone di nuovi tra gli attributi scelti dall'utente per la query.

Durante l'esecuzione della pagina *ShowAddTable.jsp* di visualizzazione dei risultati, il sistema carica, dalla query salvata nella sessione, tutti i filtri impostati in precedenza (nella pagina di composizione della query oppure salvati nel database) e li visualizza con una stringa che indica il campo su cui viene effettuato il filtro, l'operatore e il valore del filtro stesso (Figura 34).

Figura 34 - Esempio di visualizzazione di filtri

Cliccando sulla "[X]" a fianco di ogni filtro è possibile eliminare il filtro dalla query, mentre cliccando sulla label "[edit]" viene aperto un menu che permette di modificare il filtro stesso (Figura 35).

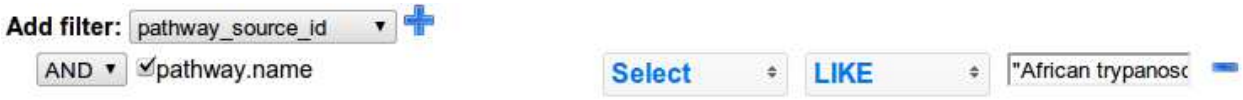


Figura 35 - Menu di aggiunta e/o modifica filtri

E' possibile inoltre aggiungere nuovi filtri selezionando dal menu a tendina il campo che si vuole filtrare e cliccando su "+". In questo modo verrà visualizzato un menu simile a quello per la modifica dei filtri che permetterà di impostare il nuovo filtro.

Deselezionando il checkbox a fianco del nome dell'attributo, il filtro in oggetto viene eliminato dalla query ma non dal form, e può essere quindi velocemente resinserito. Cliccando invece sul "-" all'estrema destra del menu il filtro viene rimosso dalla query e dal form, e sarà quindi eventualmente necessario reimpostarlo.

Cliccando sul bottone "*Search*" verrà eseguita una chiamata alla servlet *DataTableGPKB* con i nuovi filtri e quindi verrà eseguita la nuova query e verranno mostrati i suoi risultati nella tabella.

Dell'iniziale generazione della listi di filtri impostati in fase di creazione della query si occupa la funzione javascript *SetSearchFields(),* che, per ogni filtro già impostato, effettua una chiamata alla funzione *FillSearchAttributeSpan()*, che inserisce nella pagina le righe per la modifica dei filtri (prendendo i dati, con una chiamata Ajax, dalla classe *FeatureAttributesTemplate* (riproducendo quindi i menu usati per la composizione della query) tramite la servlet *FillSearchAttributeSpan)* inserendoli in un form i cui attributi verranno poi passati alla servlet *SetSearchFields* per la generazione della query con i nuovi filtri.

Sempre nella funzione javascript *FillSearchAttributeSpan()*, per ogni attributo, viene generato il suo riquadro che ne rappresenta il filtro, tramite una chiamata alla funzione *addFilterString()*.

Infine, la stessa funzione si occupa di nascondere alla vista i menu di modifica dei filtri e di mostrare solo i riquadri che li rappresentano.

Nel momento in cui l'utente decide di modificare un filtro già impostato cliccando sulla label "[edit]", viene eseguita una chiamata alla funzione *showRow()* che semplicemente mostra il menù di modifica del filtro dell'attributo selezionato. Analogamente, se l'utente decide di eliminare il filtro cliccando su "[X]", con una chiamata alla funziona *deleteRow()* viene rimosso dal form il menù di modifica dell'attributo.

L'utente può decidere di togliere temporaneamente il filtro dalla ricerca, senza però cancellarlo, deselezionando il checkbox nel menù di modifica del filtro stesso.

Se l'utente decide di impostare un nuovo filtro, seleziona dal menù a tendina (contenente tutti e soli gli attributi selezionati in fase di composizione query, filtrati e non) l'attributo da filtrare, e clicca su "+". Viene quindi generato e aggiunto al form un nuovo menù di modifica filtro, che l'utente potrà impostare.

Cliccando sul bottone "Search", tutti gli elementi del form generato vengono passati, con una chiamata ajax, alla servlet *SetSearchFields* che si occupa di generare la nuova clausola "where" della query e di salvarla nella sessione. Infine, viene rimandata in esecuzione la servlet *DataTableGPKB*, che genera la nuova query, la esegue e ne mostra i risultati.

6.5 Salvataggio, caricamento e riesecuzione delle query

E' stato implementato un sistema di salvataggio e caricamento di query da parte di ogni utente. In questa sezione si spiega nei dettagli ciò che è stato fatto.

6.5.1 Tabelle "query" e "query_attribute" in database GPDW_web

Sono state inserite nel database ***GPDW_Web*** le tabelle *query* e *query_attribute*, in modo che fosse possibile salvare al suo interno le query predefinite e le query che ogni utente desidera salvare e riutilizzare.

In Figura 36 si riporta lo schema ER di questa parte del database.

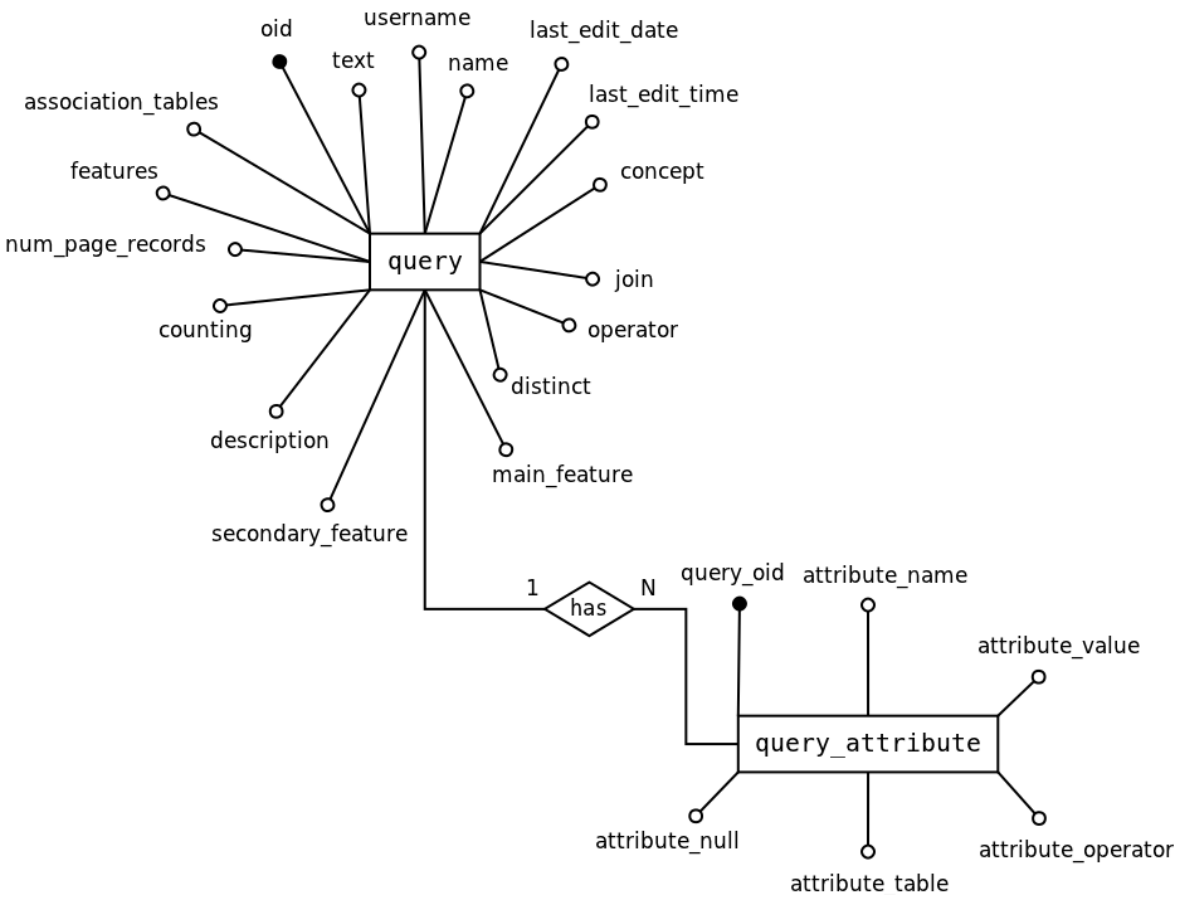


Figura 36 - Schema ER delle tabelle query e query_attribute

Nella tabella **query** vengono salvati i dati generali di ogni query. Possiede i seguenti attributi:

- ***oid***: un numero progressivo unico che identifica la query. E' la chiave primaria di questa tabella

- ***text***: il testo della query. Compilato nel caso di query executable, null nel caso di query loadable
- ***username***: il nome dell'utente proprietario di questa query
- ***name***: nome identificativo della query. La coppia (name,username) deve essere unica nella tabella.
- ***last_edit_date***: giorno dell'ultima modifica
- ***last_edit_time***: ora dell'ultima modifica
- ***concept***: indica su quali feature della query effettuare una query concept (vedi sezione 6.1.4)
- ***join***: indica il tipo di join (inner o outer) della query
- ***operator***: indica il tipo di "filter" (AND o OR) della query
- ***distinct***: indica se applicare la clausola DISTINCT oppure no
- ***main_feature***: il nome della feature principale della query
- ***secondary_feature***: il nome della feature secondaria della query (null nel caso di singola feature)
- ***description***: la descrizione, inserita dall'utente, della query
- ***counting***: indica la modalità di counting della query (exact o estimated)
- ***num_page_records***: indica quanti risultati per pagina mostrare all'utente in fase di visualizzazione dei risultati.
- ***features:*** contiene la lista delle features selezionate dall'utente per questa query. Possono essere al massimo due nel caso single o pairwise feature, mentre possono essere molte nel caso di multiple feature search.
- ***association_tables***: contiene la lista delle tabelle di associazione selezionate dall'utente per questa query.

La tabella **query_attributes**, invece, si occupa di salvare tutti gli attributi di ogni query. E' composta dai seguenti attributi:

- ***query_oid***: identificativo numerico della query a cui si riferisce l'attributo in questione. E' in relazione con un vincolo di chiave esterna con l'attributo *oid* della tabella *query*
- ***attribute_name***: il nome dell'attributo
- ***attribute_value***: il valore dell'attributo
- ***attribute_operator***: l'operatore della clausola where dell'attributo (se presente)
- ***attribute_table***: la tabella dell'attributo

- ***attribute_null***: indica se è stato selezionato null o not null nella selezione delle condizioni di questo attributo.

Salvataggio di una query

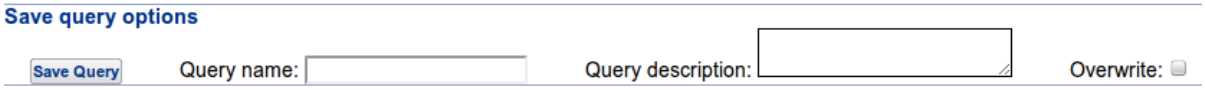


Figura 37 - Menu di salvataggio di una query

In fase di composizione di una query o di visualizzazione dei risultati, l'utente può decidere di salvare la sua query utilizzando il menu *"Save query options"* (Figura 37), indicando un nome identificativo della query e una breve descrizione. Se il nome esiste già e l'utente ha intenzione di sovrascrivere la query preesistente, dovrà spuntare il checkbox "overwrite".

Quando viene salvata una query, viene effettuata una chiamata AJAX alla servlet *SaveQuery*, che si occupa (se non è già presente) di salvare la query nella sessione (tramite la funzione *storeQuery()*) e in seguito di salvarla nel database compilando le tabelle precedentemente descritte, prendendo i dati dalla request generata dal form della pagina *selectAttributes.jsp* compilato dall'utente.

Salvataggio di una query temporanea nella sessione

Durante ogni cambiamento o ricaricamento di una pagina durante la composizione della query o la visualizzazione dei suoi risultati, la query in composizione viene salvata temporaneamente nella sessione dell'utente, per poter fare in modo che questa si possa visualizzare nel form di composizione in modo corretto e si possa salvare nel database anche in fase di visualizzazione dei risultati (cioè quando il form di composizione non è più presente e non è più possibile inviare i dati della request del form alla servlet *SaveQuery*).

Di questo salvataggio nella sessione si occupa la funzione *SaveQuery.storeQuery()* che, analizzando la request proveniente dal form compilato dall'utente nella pagina *selectAttributes.jsp*, compila un'istanza della classe *QuerySQL*, che contiene tutti i dati necessari per poter identificare la query.

In questo modo, quando viene ricaricata la pagina di composizione della query, se è presente una query nella sessione (e ciò avviene quando l'utente ha già cominciato la composizione di una query, ad es.: quando l'utente seleziona una feature e poi estende la query a una seconda

feature), il form di composizione viene compilato con i valori precedentemente inseriti dall'utente (e salvati nella sessione) permettendo così la continuazione della modifica della query in composizione.

Di questa compilazione con i valori salvati nella sessione se ne occupa la funzione *FeatureAttributesTemplate.getTableAttributeRow(),* che compila il form di composizione con gli attributi delle tabelle con i valori salvati nella query salvata a sua volta nella sessione.

Caricamento di una query

Figura 38 - Interfaccia di gestione delle query

Durante tutte le fasi di ricerca dei dati nell'applicazione (scelta di una feature, selezione degli attributi e visualizzazione dati) è possibile caricare una query predefinita o precedentemente salvata dall'utente.

Questo avviene selezionando una query dal menu a tendina "Predefined queries" (Figura 40). A seconda del tipo di query, sarà possibile caricarla (mostrando il suo form di composizione con i menu già compilati con i valori salvati) oppure eseguirla direttamente. I tipi di query sono i seguenti:

- ***Loadable query***: queste query possono essere caricate nel form di composizione per essere eseguite di nuovo o ulteriormente modificate. Tutte le query salvate da un utente sono di tipo loadable.
- ***Executable query***: queste query possono solo essere eseguite direttamente, senza passare dal form di composizione. Non possiedono attributi salvati nella tabella ***query_attributes***, e il loro testo è salvato nell'attributo *text* della tabella ***query***.
- ***Both***: queste query possono essere sia caricate nel form di composizione che essere eseguite direttamente. Ogni query "*loadable*", dopo che è stata eseguita almeno una volta, diventa anche "*executable*".

Per ognuno di questi tipi, è presente il corrispettivo "***multiple***": se una query coinvolge più di due features, allora sarà caricabile soltanto nel menù grafico di composizione della query, e non in quello visuale.

Nel menu "Predefined queries", per ogni utente diverso dall'utente *guest*, è possibile anche cancellare le query precedentemente salvate tramite il bottone *"Delete query"*.

Nel momento in cui l'utente seleziona una query dal menu a tendina, a seconda del diverso tipo della query vengono attivati i bottoni "*Load query*", "*Load query in visual search*", "*Execute query*".

Entambi i bottoni eseguono la servlet *LoadQuery* la quale, a seconda dell'azione richiesta dall'utente, esegue le seguenti azioni:

- Caso "***Load query***": il sistema carica la query dal database (*DBQueryManager.loadQuery()*) e la carica nella sessione. Dopodichè esegue la servlet *GetFeatureAttributes* che si occupa di generare la pagina di composizione della query (*selectAttributes.jsp*) con i dati caricati nella sessione (vedi sezione 5.2.1).
- Caso "***Load query in visual search***": analogo al caso precedente, soltanto che la query viene mostrata nella modalità grafica di composizione (*visualSearch.jsp*), tramite il grafo delle features.
- Caso "***Executable query***": il sistema carica il testo completo della query salvato nel database (*DBQueryManager.loadExecutableQuery()*) e lo inserisce nella sessione. Dopodichè esegue direttamente la pagina *ShowAddTable.jsp* di visualizzazione dei risultati. Siccome non si passa per la pagina di composizione della query, si rende necessario impostare alcuni parametri (che verranno caricati nella sessione) per la corretta esecuzione della query e visualizzazione dei risultati:
    - viene generato il testo della query che verrà messo in output agli utenti non di tipo guest
    - viene generata l'intestazione della tabella della visualizzazione dei dati
    - viene determinata la modalità di count
    - viene determinato il numero di risultati da visualizzare per ogni pagina

7 Performance e testing

7.1 Peformance

In questa sezione vengono descritte le performance del sistema, in termini di tempi di esecuzione delle query, di generazione delle diverse tipologie di interfaccia (visuale e grafica) e di utilizzo della memoria del webserver.

7.1.1 Tempi di esecuzione delle query

Prima di esporre i test effettuati, sono necessarie alcune premesse.
Sono state composte ed eseguite alcune query significative, che coinvolgessero una sola feature, due feature e feature multiple. Sono stati rilevati i loro tempi di esecuzione, le tuple restituite da ciascuna query e il tempo in cui il conteggio di queste tuple è stato ottenuto.
Sono state scelte query il cui percorso di join generato dall'algoritmo sviluppato in questo lavoro di Tesi fosse più corto rispetto a quello generato seguendo i vincoli di chiave esterna presenti nel GPDW. Sono stati così riportati confronti tra i diversi tempi di esecuzione delle due tipologie di query.

Come dettagliatamente illustrato nella sezione 2.1, nel GPDW, tra le diverse "feature", è presente una distinzione: ci sono feature che sono "entità biomolecolari" ("gene", "protein","dna_sequence","transcript") e feature che sono "feature biomediche" (tutte le rimanenti). I dati presenti nel database per le entità biomolecolari sono molti: ognuna delle loro tabelle principali di feature, infatti, possiede milioni di tuple (la più piccola, dna_sequence, a oggi posside circa 4 milioni di tuple).
Di conseguenza, per i test di performance, è necessario distinguere i casi in cui nella query sia presente una (o più) entità biomolecolari rispetto alle feature biomediche.

PostgreSQL, inoltre, possiede un sistema di caching delle tabelle coinvolte nelle query che permette un'esecuzione più veloce delle query se le tabelle che questa interroga sono già state interrogate precedentemente e quindi si trovano in cache.

Per partire da una situazione univoca e per uniformità di testing, si è scelto di riportare i tempi di esecuzione della seconda esecuzione di una query, dopo che i suoi dati sono già stati messi in cache. Per ogni query, è stato indicato anche il tempo impiegato dalla sua prima esecuzione.

Singola feature

I primi test sono stati effettuati sulla ricerca all'interno di un solo modulo. In particolare, si è scelto di effettuare alcune query d'esempio su una feature del GPDW. E' stata scelta la feature "biological_function_feature", in quanto il suo modulo comprende tabelle di sorgente e tabelle addizionali di sorgente e tabelle addizionali di feature, permettendo così la composizione di query "articolate". La tabella principale di feature di "biological_function_feature" possiede 36.963 tuple.

La prima query effettuata ha compreso la tabella principale di feature ("biological_function_feature"), una tabella addizionale di feature ("biological_function_feature_synonym") e una tabella addizionale di sorgente ("go_biological_process_alternative_id"). Gli attributi selezionati sono stati quelli di default.
E' stata scelta questa query in quanto il suo percorso di join naturale, cioè seguendo le chiavi esterne fra le tabelle, includerebbe anche la tabella di sorgente relativa alla tabella addizionale di sorgente inserita. L'algoritmo, però, sfruttando la struttura del GPDW, non la include, riducendo così il numero di join effettuati dalla query.
Il testo della query è riportato in Tabella 12.

```
SELECT
biological_function_feature_synonym.synonym AS
biological_function_feature_synonym_synonym,
go_biological_process_alternative_id.alternative_id AS
go_biological_process_alternative_id_alternative_id
FROM public.biological_function_feature
LEFT JOIN public.go_biological_process_alternative_id ON
public.biological_function_feature.biological_function_feature_oid=pub
lic.go_biological_process_alternative_id.go_biological_process_oid
LEFT JOIN public.biological_function_feature_synonym ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature_synonym.biological_function_feature
_oid
ORDER BY biological_function_feature_synonym.synonym ASC LIMIT 20
OFFSET 0
```

Tabella 12 - Testo della prima query di test delle performance. Singola feature.

Il tempo di esecuzione della query è stato di 116 millisecondi (311 ms la prima esecuzione senza dati in cache), il numero esatto di tuple restituite è stato 91.224 (conteggio ottenuto in 88 millisecondi).

Si riporta in Tabella 13 la clausola FROM della query corrispondente al caso precedente che sarebbe stata generata seguendo tutti i percorsi di join indicati dai vincoli di chiave esterna nel GPDW. In grassetto il join aggiunto rispetto alla query in Tabella 12.

```
FROM public.biological_function_feature
LEFT JOIN public.biological_function_feature_synonym ON
public.biological_function_feature.biological_function_feature_oid=
public.biological_function_feature_synonym.biological_function_feature
_oid
LEFT JOIN public.go_biological_process ON
public.biological_function_feature.biological_function_feature_oid=pub
lic.go_biological_process.go_biological_process_oid
LEFT JOIN public.go_biological_process_alternative_id ON
public.go_biological_process.go_biological_process_oid=public.go_biolo
gical_process_alternative_id.go_biological_process_oid
```

Tabella 13 - Clausola FROM della query di test con il percorso di join generato seguendo i vincoli di chiave esterna

Il tempo di esecuzione della query (eseguita con il client pgAdmin, senza le clausole ORDER BY e LIMIT per testare l'esecuzione della query con la sola composizione dei join) è stato di 5.805 millisecondi. La query precedente, generata dall'algoritmo sviluppato in questo lavoro di Tesi, eseguita in pgAdmin (anch'essa senza le clausole ORDER BY e LIMIT) ha mostrato un equivalente tempo di esecuzione: 5756 millisecondi. Il numero di tuple ritornate, 91.224, è stato lo stesso in entrambi i casi.

Doppia feature

Si considera una query che coinvolge un'associazione tra una entità biomolecolare ("gene") e una feature biomedica ("gene_expression_feature"). Il numero di tuple della tabella principale di feature di "gene" è molto elevato: 10.066.306 tuple. Il numero di tuple della tabella di "gene_expression_feature", invece, è di 1.065 tuple, molto inferiore rispetto a quello di "gene". La query generata con i soli attributi selezionati di default è mostrata in Tabella 14.

```
SELECT
gene.source_id AS gene_source_id,
gene.source_name AS gene_source_name,
gene.symbol AS gene_symbol,
gene.taxonomy_id AS gene_taxonomy_id,
gene_expression_feature.source_id AS
gene_expression_feature_source_id,
gene_expression_feature.source_name AS
gene_expression_feature_source_name,
gene_expression_feature.feature_type AS
gene_expression_feature_feature_type,
gene_expression_feature.name AS gene_expression_feature_name,
gene2gene_expression_feature.association_type AS
gene2gene_expression_feature_association_type
FROM public.gene
INNER JOIN public.gene2gene_expression_feature ON
```

```
public.gene.gene_oid=public.gene2gene_expression_feature.gene_oid
INNER JOIN public.gene_expression_feature ON
public.gene_expression_feature.gene_expression_feature_oid=public.gene
2gene_expression_feature.gene_expression_feature_oid
ORDER BY gene.source_id ASC LIMIT 20 OFFSET 0
```

Tabella 14 - Testo della seconda query di test delle performance. Doppia feature.

Il tempo di esecuzione della query è stato di 4 millisecondi (364 ms la prima esecuzione) e il numero esatto di tuple ritornate è stato 1.893.188. La query di conteggio è stata eseguita in 11.249 millisecondi.

E' stata poi rieseguita la stessa query senza i campi source_id di entrambe le feature. E' interessante rilevare come questa abbia un tempo di esecuzione molto più alto: 12.997 millisecondi (circa 3.000 volte più lento). Questo è dovuto al fatto che, in questo secondo caso, nella clausola ORDER BY della query non fosse più presente il campo gene.source_id, ma invece ci fosse il campo gene.symbol. Il campo gene.source_id, infatti, è indicizzato e questo velocizza di molto l'operazione di ordinamento. Lo si può notare osservando il risultato dell'operazione SQL EXPLAIN ANALYZE su entrambe le query. Questa operazione descrive le operazioni seguite dal DBMS per eseguire la query.

Il risultato del comando EXPLAIN ANALYZE sulla prima query (con gene.source_id nella clausola ORDER BY) è il mostrato in Tabella 15.

```
 Limit  (cost=0.00..399.74 rows=20 width=79) (actual time=0.088..0.647
rows=20 loops=1)
   ->  Nested Loop  (cost=0.00..37839403.41 rows=1893188 width=79)
(actual time=0.086..0.624 rows=20 loops=1)
         ->  Nested Loop  (cost=0.00..37308844.14 rows=1893188
width=47) (actual time=0.064..0.374 rows=20 loops=1)
               -> Index Scan using gene_source_id_source_name_key on
gene  (cost=0.00..899960.09 rows=10066449 width=43) (actual
time=0.040..0.040 rows=1 loops=1)
               ->  Index Scan using
integr_gene2gene_expression_feature_idx1 on
gene2gene_expression_feature  (cost=0.00..2.30 rows=105 width=20)
(actual time=0.014..0.296 rows=20 loops=1)
                     Index Cond: (gene_oid = gene.gene_oid)
         ->  Index Scan using gene_expression_feature_pkey on
gene_expression_feature  (cost=0.00..0.27 rows=1 width=48) (actual
time=0.008..0.009 rows=1 loops=20)
               Index Cond: (gene_expression_feature_oid =
gene2gene_expression_feature.gene_expression_feature_oid)
```

Tabella 15 - Risultato del comando EXPLAIN ANALYZE sulla query con campo source_id in clausola ORDER BY

Il risultato del comando EXPLAIN ANALYZE sulla seconda query (con gene.symbol nella clausola ORDER BY) è mostrato in Tabella 16.

```
Limit  (cost=664504.03..664504.08 rows=20 width=58) (actual
time=29827.259..29827.292 rows=20 loops=1)
   -> Sort  (cost=664504.03..669237.00 rows=1893188 width=58) (actual
time=29827.255..29827.265 rows=20 loops=1)
        Sort Key: gene.symbol
```

```
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Hash Join  (cost=422970.07..614126.98 rows=1893188
width=58) (actual time=17869.268..27962.488 rows=1893188 loops=1)
               Hash Cond:
(gene2gene_expression_feature.gene_expression_feature_oid =
gene_expression_feature.gene_expression_feature_oid)
               ->  Hash Join  (cost=422929.10..588054.68 rows=1893188
width=26) (actual time=17867.348..25287.316 rows=1893188 loops=1)
                     Hash Cond: (gene2gene_expression_feature.gene_oid
= gene.gene_oid)
                     ->  Seq Scan on gene2gene_expression_feature
(cost=0.00..36625.88 rows=1893188 width=20) (actual
time=0.007..1438.354 rows=1893188 loops=1)
                     ->  Hash  (cost=238114.49..238114.49
rows=10066449 width=22) (actual time=17857.392..17857.392
rows=10066306 loops=1)
                           Buckets: 1048576  Batches: 2  Memory Usage:
283824kB
                           ->  Seq Scan on gene  (cost=0.00..238114.49
rows=10066449 width=22) (actual time=0.018..8629.924 rows=10066306
loops=1)
               ->  Hash  (cost=27.65..27.65 rows=1065 width=48)
(actual time=1.902..1.902 rows=1065 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 88kB
                     ->  Seq Scan on gene_expression_feature
(cost=0.00..27.65 rows=1065 width=48) (actual time=0.011..0.914
rows=1065 loops=1)
```

Tabella 16 - Risultato del comando EXPLAIN ANALYZE sulla query senza campo source_id in clausola ORDER BY

Come si può notare, nella seconda query viene effettuata un'operazione di SORT sui dati ottenuti, che sono molti (`rows=1893188`). Questa operazione, molto onerosa, invece non avviene nella prima query, dove viene effettuata fin da subito un'operazione di INDEX SCAN sulla tabella gene: questo significa che i dati ottenuti dopo aver effettuato i join risultano già ordinati secondo il campo gene.source_id (poiché questo è un campo indicizzato), e quindi non è necessario effettuare nessun ordinamento.

L'assenza dell'operazione di ordinamento si può notare anche dalla rappresentazione grafica del comando EXPLAIN effettuata da pgAdmin: in Figura 39 è mostrata la query con il campo gene.source_id, mentre in Figura 40 è mostrata la query senza il campo gene.source_id. Nel secondo caso è presente l'operazione di SORT, assente nella prima figura.

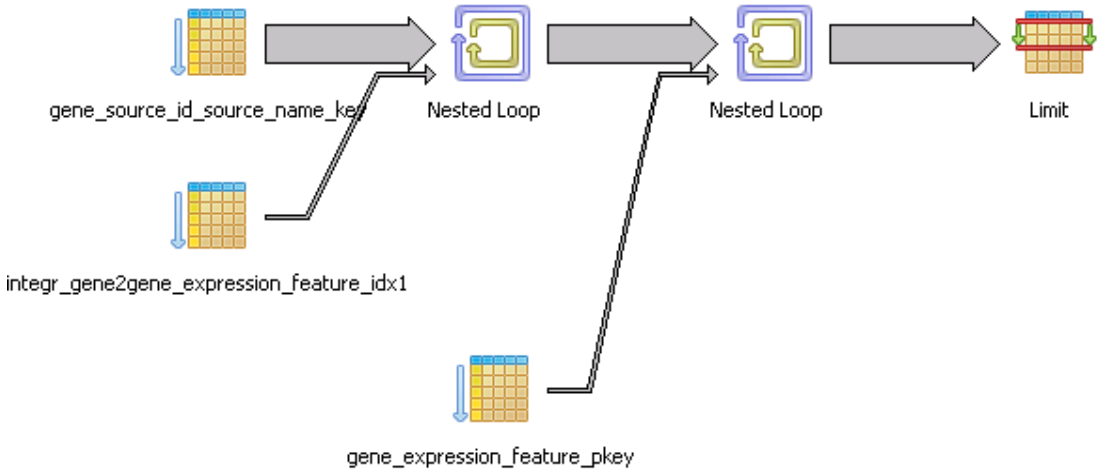


Figura 39 - Rappresentazione grafica del comando EXPLAIN sulla query con ordinamento su campo indicizzato

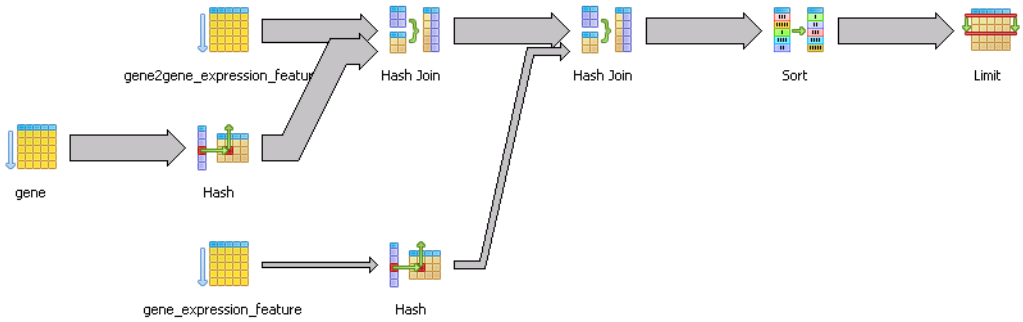


Figura 40 - Rappresentazione grafica del comando EXPLAIN sulla query con ordinamento su campo non indicizzato

Questa grande differenza nei tempi di esecuzione di una query con ordinamento su un campo indicizzato rispetto a una su un campo non indicizzato risulta interessante in termini di performance del data warehouse, in quanto potrebbe essere utile discturere se indicizzare nuovi campi oltre a quelli su cui è già stata effettuata questa operazione nel GPDW.

Eseguendo poi la query con gli attributi di default aggiungendovi due filtri (gene_expression_feature.feature_type = “cellular_type” e gene_expression_feature.name IN %FE%), la query è stata eseguita in 39.501 millisecondi, e ha restituito 0 tuple (conteggio ottenuto in 349 millisecondi). In questo caso l’impostazione di filtri onerosi come LIKE ha contribuito ad allungare di molto il tempo di esecuzione della query (di quasi 10.000 volte).

E’ stata poi eseguita una query tra le feature “gene” e “biological_funcion_feature”. In particolare, è stata generata una query che coinvolgesse una tabella addizionale di feature di biological_function_feature (“biological_function_feature_alternative_id”) e una tabella

addizionale di sorgente di gene (“omim_gene_method”). Anche in questo caso, il percorso di join generato dall’algoritmo di composizione dalla query sviluppato in questa Tesi è molto più breve rispetto a quello indicato dalle chiavi esterne.

La query generata è mostrata in Tabella 17.

```
SELECT
omim_gene_method.method AS omim_gene_method_method,
biological_function_feature_alternative_id.alternative_id AS
biological_function_feature_alternative_id_alternative_id
FROM public.gene2biological_function_feature
INNER JOIN public.biological_function_feature_alternative_id ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.biological_function_feature_alternative_id.biological_functio
n_feature_oid
INNER JOIN public.omim_gene_method ON
public.gene2biological_function_feature.gene_oid=public.omim_gene_meth
od.omim_gene_oid
ORDER BY omim_gene_method.method ASC LIMIT 20 OFFSET 0
```

Tabella 17 - Testo della terza query di test delle performance.

Il tempo di esecuzione della query è stato di 741 millisecondi (797 ms la prima esecuzione) e il numero esatto di tuple ritornate è stato 1.228 (conteggio ottenuto in 757 millisecondi).

Come per il caso singola feature, in Tabella 18 si riporta il testo della clausola FROM che sarebbe stata generata seguendo il percorso indicato dalle chiavi esterne nel GPDW. In grassetto sono mostrati i join aggiunti nella query.

```
FROM public.gene2biological_function_feature
LEFT JOIN public.gene ON
public.gene2biological_function_feature.gene_oid=public.gene.gene_oid
LEFT JOIN public.biological_function_feature ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.biological_function_feature.biological_function_feature_oid
LEFT JOIN public.omim_gene ON
public.gene.gene_oid=public.omim_gene.omim_gene_oid
LEFT JOIN public.biological_function_feature_alternative_id ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.biological_function_feature_alternative_id.biological_functio
n_feature_oid
LEFT JOIN public.omim_gene_method ON
public.gene2biological_function_feature.gene_oid=public.omim_gene_meth
od.omim_gene_oid
```

Tabella 18 - Clausola FROM della query generata seguendo il percorso di join indicato dalle chiavi esterne.

Questa query, eseguita in pgAdmin (senza clausole di ORDER BY e LIMIT), è stata eseguita in 43.700 millisecondi. La query in Tabella 17, modificata togliendo le clausole ORDER BY e LIMIT e modificando gli INNER JOIN in LEFT JOIN, eseguita in pgAdmin è stata eseguita in 43.093 ms, un tempo in linea con il precedente. Entrambe le query hanno restituito lo stesso numero di tuple: 1.763.441.

Feature multiple

Si considera una query che coinvolge tre feature, di cui solo una è una entità biomolecolare ("gene"), mentre le altre sono feature biomediche ("enzyme", la cui tabella principale di feature ha 5.190 tuple e "protein_fam_dom", la cui tabella principale di feature possiede 19.522 tuple). Le associazioni considerate sono state quella tra gene ed enzyme ("gene2enzyme") e tra enzyme e protein_fam_dom ("enzyme2protein_fam_dom").

La query, eseguita con gli attributi di default, è riportata in Tabella 19.

```
SELECT
gene.source_id AS gene_source_id,
gene.source_name AS gene_source_name,
gene.symbol AS gene_symbol,
gene.taxonomy_id AS gene_taxonomy_id,
enzyme.source_id AS enzyme_source_id,
enzyme.source_name AS enzyme_source_name,
enzyme.name AS enzyme_name,
protein_fam_dom.source_id AS protein_fam_dom_source_id,
protein_fam_dom.source_name AS protein_fam_dom_source_name,
protein_fam_dom.name AS protein_fam_dom_name
FROM public.gene
INNER JOIN public.gene2enzyme ON
public.gene.gene_oid=public.gene2enzyme.gene_oid
INNER JOIN public.enzyme ON
public.enzyme.enzyme_oid=public.gene2enzyme.enzyme_oid
INNER JOIN public.enzyme2protein_fam_dom ON
public.enzyme.enzyme_oid=public.enzyme2protein_fam_dom.enzyme_oid
INNER JOIN public.protein_fam_dom ON
public.protein_fam_dom.protein_fam_dom_oid=public.enzyme2protein_fam_d
om.protein_fam_dom_oid
ORDER BY gene.source_id ASC LIMIT 20 OFFSET 0
```

Tabella 19 - Testo della quarta query di test delle performance. Feature multiple.

E' stata eseguita in 14 millisecondi (16 ms la prima esecuzione) e ha restituito 201.820 tuple (conteggio ottenuto in 992 millisecondi).

Applicando dei filtri a questa query (gene.taxonomy_id = "Danio rerio" ed enzyme.enzyme_name LIKE "DNA%"), si ottengono 96 tuple (conteggio ottenuto in 506 millisecondi) con un tempo di esecuzione di 499 millisecondi. Anche in questo caso i filtri hanno rallentato l'esecuzione della query, anche se in modo minore rispetto al caso descritto nella sezione precedente (query con filtri circa 35 volte più lenta). Ciò è dovuto al fatto che i filtri applicati dalla query hanno permesso di ridurre di molto le tuple esaminate (`rows=17` su enzyme): ciò ha permesso di ridurre i tempi di svolgimento dei join.

Lo si può vedere esaminando il risultato del comando EXPLAIN ANALYZE sulla query eseguita, in Tabella 20.

```
Limit  (cost=4892.74..4892.74 rows=2 width=627) (actual
time=719.153..719.183 rows=20 loops=1)
  ->  Sort  (cost=4892.74..4892.74 rows=2 width=627) (actual
time=719.150..719.161 rows=20 loops=1)
        Sort Key: protein_fam_dom.source_id
        Sort Method: top-N heapsort  Memory: 28kB
        ->  Nested Loop  (cost=9.53..4892.73 rows=2 width=627) (actual
time=380.271..718.594 rows=96 loops=1)
              ->  Nested Loop  (cost=9.53..4890.54 rows=2 width=97)
(actual time=380.252..717.849 rows=96 loops=1)
                    ->  Nested Loop  (cost=9.53..3390.81 rows=395
width=70) (actual time=3.489..206.585 rows=61660 loops=1)
                          ->  Nested Loop  (cost=0.00..260.39 rows=8
width=78) (actual time=3.429..10.033 rows=31 loops=1)
                                ->  Seq Scan on enzyme
(cost=0.00..172.82 rows=27 width=62) (actual time=2.731..9.595 rows=17
loops=1)
                                      Filter: ((upper((name)::text) ~~
'DNA%'::text))
                                ->  Index Scan using
integr_enzyme2protein_fam_dom_idx1 on enzyme2protein_fam_dom
(cost=0.00..3.23 rows=1 width=16) (actual time=0.009..0.020 rows=2
loops=17)
                                      Index Cond: (enzyme_oid =
enzyme.enzyme_oid)
                          ->  Bitmap Heap Scan on gene2enzyme
(cost=9.53..389.40 rows=152 width=16) (actual time=0.542..3.924
rows=1989 loops=31)
                                Recheck Cond: (enzyme_oid =
enzyme.enzyme_oid)
                                ->  Bitmap Index Scan on
gene2enzyme_6_idx  (cost=0.00..9.49 rows=152 width=0) (actual
time=0.365..0.365 rows=1989 loops=31)
                                      Index Cond: (enzyme_oid =
enzyme.enzyme_oid)
                    ->  Index Scan using gene_pkey on gene
(cost=0.00..3.78 rows=1 width=43) (actual time=0.007..0.007 rows=0
loops=61660)
                          Index Cond: (gene_oid =
gene2enzyme.gene_oid)
                          Filter: (taxonomy_id = 7955)
              ->  Index Scan using protein_fam_dom_pkey on
protein_fam_dom  (cost=0.00..1.08 rows=1 width=546) (actual
time=0.004..0.005 rows=1 loops=96)
                    Index Cond: (protein_fam_dom_oid =
enzyme2protein_fam_dom.protein_fam_dom_oid)
```

Tabella 20 - Risultato del comando EXPLAIN ANALYZE sulla query di test per feature multipla con filtri applicati

E' stata eseguita, infine, una query comprendente tre features ("biological_function_feature", "gene" e "genetic_disorder"), includendo una tabella addizionale di sorgente di gene ("entrez_gene_synonym"), una tabella addizionale di sorgente di "biological_function_feature" ("go_molecular_process") e una tabella addizionale di sorgente di "genetic_disorder" ("omim_disorder_text"). In Tabella 21 si riporta la clausola FROM di questa query

```
FROM public.gene2biological_function_feature
LEFT JOIN public.go_molecular_function_subset ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.go_molecular_function_subset.go_molecular_function_oid
LEFT JOIN public.entrez_gene_synonym ON
public.gene2biological_function_feature.gene_oid=public.entrez_gene_sy
nonym.entrez_gene_oid
LEFT JOIN public.gene2genetic_disorder ON
public.entrez_gene_synonym.entrez_gene_oid=public.gene2genetic_disorde
r.gene_oid
LEFT JOIN public.omim_disorder_text ON
public.omim_disorder_text.omim_disorder_oid=public.gene2genetic_disord
er.genetic_disorder_oid
```

Tabella 21 - Clausola FROM di una query su tre features con solo loro tabelle addizionali di sorgente

Questa query, eseguita in pgAdmin, è stata eseguita in 353.612 ms.

Analogamente a quanto svolto per le query precedenti, è stata poi eseguita la stessa query (di cui in Tabella 22 si riporta la sola clausola FROM) con il percorso di join generato seguendo i vincoli di chiave esterna delle sue tabelle. In grassetto i join aggiunti rispetto alla query precedente.

```
FROM public.gene2biological_function_feature
LEFT JOIN public.gene ON
public.gene2biological_function_feature.gene_oid=public.gene.gene_oid
LEFT JOIN public.entrez_gene ON
public.gene.gene_oid=public.entrez_gene.entrez_gene_oid
LEFT JOIN public.biological_function_feature ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.biological_function_feature.biological_function_feature_oid
LEFT JOIN public.go_molecular_function ON
public.biological_function_feature.biological_function_feature_oid=pub
lic.go_molecular_function.go_molecular_function_oid
LEFT JOIN public.go_molecular_function_subset ON
public.gene2biological_function_feature.biological_function_feature_oi
d=public.go_molecular_function_subset.go_molecular_function_oid
LEFT JOIN public.entrez_gene_synonym ON
public.gene2biological_function_feature.gene_oid=public.entrez_gene_sy
nonym.entrez_gene_oid
LEFT JOIN public.gene2genetic_disorder ON
public.entrez_gene_synonym.entrez_gene_oid=public.gene2genetic_disorde
r.gene_oid
LEFT JOIN public.genetic_disorder ON
public.gene2genetic_disorder.genetic_disorder_oid=public.genetic_disor
der.genetic_disorder_oid
LEFT JOIN public.omim_disorder ON
public.genetic_disorder.genetic_disorder_oid=public.omim_disorder.omim
_disorder_oid
LEFT JOIN public.omim_disorder_text ON
public.omim_disorder_text.omim_disorder_oid=public.gene2genetic_disord
er.genetic_disorder_oid
```

Tabella 22 - Clausola FROM della query comprendente tre features generata seguendo il percorso indicato dai vincoli di chiave esterna.

Questa query è stata eseguita in 359.451 ms, un tempo maggiore rispetto alla query generata dall'algoritmo sviluppato in questo lavoro di Tesi.

Entrambe le query hanno restituito 10.647.473 tuple.

7.1.2 Discussione sui tempi di esecuzione delle query

Come riportato nella sezione precedente, i tempi di esecuzione delle query generate originariamente dall'applicazione sono inferiori al secondo anche in casi in cui sono state esaminate milioni di tuple: ciò dimostra la bontà della formulazione delle query e delle strategia di ricerca all'interno del GPDW adottate dal DBMS.

Non si è riscontrata una grande differenza, in termini di performance, tra le query il cui percorso di join seguiva le indicazioni fornite dai vincoli di chiave esterna rispetto alle query generate dall'algoritmo descritto nella sezione 6.1. I tempi di esecuzione sono risultati molto simili tra loro. Solo nell'ultima query, più complessa e onerosa, si è avuto un miglioramento tangibile di 6 secondi, anche se percentualmente piccolo, da parte della query generata con l'algoritmo oggetto di questo lavoro di Tesi. Si può supporre che, aumentando il livello di complessità della query, aumenti anche il beneficio dato dall'algoritmo stesso.

Questo dimostra, inoltre, la bontà della progettazione e implementazione del GPDW e degli indici e vincoli di chiave e chiave esterna posti, nonché del DBMS sottostante che, indipendentemente dalla formulazione della query, genera un esecuzione ottimizzata dell'interrogazione. Ciò non toglie che, con un DBMS differente e meno efficiente, le query generate dall'algoritmo descritto in questo lavoro di Tesi, effettuando meno join rispetto a quelle con percorso di join "completo", risultino formulate in maniera più efficiente e abbiano quindi maggiori possibilità di essere eseguite in tempo minore rispetto alla loro controparte "completa".

Le query composte con l'algoritmo descritto nella sezione 6.1, infine, risultano comunque essere di miglior comprensione nella loro lettura, essendo composte da meno clausole di join.

7.1.3 Tempi di generazione dell'interfaccia visuale

Questo lavoro di Tesi ha sviluppato una interfaccia visuale per la composizione delle query nel GPDW, descritta nella sezione 6.2.

In questa sezione si esaminano i tempi di generazione della pagina di composizione "visuale" della query, e si discute il valore aggiunto che questa interfaccia ha relativamente al maggior tempo di composizione rispetto a una definizione testuale SQL "standard" della query.

L'interfaccia visuale sviluppata permette la composizione di query che coinvolgono una feature soltanto oppure due features. Si distinguono, quindi, i due casi.

Nelle tabelle riportate in questa sezione, per "Tempo di esecuzione servlet" si intende il tempo di esecuzione totale della servlet *GetFeatureAttributes*, che si occupa di prelevare dal database i nomi delle tabelle relative alla feature richiesta e di mandare in esecuzione la pagina selectAttributes.jsp. Per "Tempo di esecuzione Javascript", invece, si intende il tempo di esecuzione totale della pagina selectAttributes.jsp, la quale comprende anche le chiamate Ajax generate per popolare le search options con i valori corretti e i menù degli attributi di ciascuna tabella (per i dettagli, vedi sezione 6.2).

Composizione singola feature

In Tabella 23 sono riportati i tempi in millisecondi di generazione della pagina di composizione delle query, in modalità visuale, per ogni feature (singola feature).

Feature	Tempo esecuzione servlet	Tempo generazione JavaScript	Tempo totale
Biological Function Feature	517	416	933
Clinical Synopsis	155	423	578
Dna Sequence	65	324	389
Enzyme	134	351	485
Gene	334	400	734
Gene Expression Feature	405	352	757
Genetic Disorder	226	491	717
Pathway	114	558	672
Protein	97	324	421
Protein Fam Dom	60	589	649
Small Molec	69	294	363
Transcript	58	329	387
Media	**186,17**	**404,25**	**590,42**
Deviazione standard	**153,74**	**96,23**	**182,52**

Tabella 23 - Tempi in millisecondi di caricamento della pagina di composizione query in modalità "visuale", singola feature

La media del tempo di esecuzione della parte "servlet" è molto inferiore rispetto alla parte "Javascript".

In ogni caso, la media totale di generazione di queste pagine è di poco superiore al mezzo secondo.

Composizione doppia feature

In Tabella 24 si riportano i tempi in millisecondi di generazione della pagina di composizione di una query che coinvolge due feature. Si riportano alcuni casi significativi, cioè associazioni tra le feature che hanno riportato tempi di composizione più lenti nel caso di singola feature.

Coppia di feature	Tempo esecuzione servlet	Tempo generazione JavaScript	Tempo totale
Biological Function Feature e Gene	884	1.098	1.982
Biological Function Feature e Pathway	702	1.298	2.000
Biological Function Feature e Protein	812	1.286	2.098
Gene e Gene Expression Feature	741	1.276	2.017
Gene e Genetic Disorder	572	1.375	1.947
Media	**742,20**	**1.266,60**	**2.008,80**
Deviazione Standard	**117,86**	**102,01**	**56,21**

Tabella 24 - Tempi in millisecondi di caricamento della pagina di composizione query in modalità "visuale", doppia feature

In questo caso la media dei tempi totali di esecuzione è di circa due secondi.

Caricamento di una query predefinita

Il tempo impiegato per la generazione della pagina di composizione della query in modalità “visuale” nel caso di caricamento di query predefinita è molto variabile, e dipende dal numero di tabelle e di attributi che la query caricata coinvolge.

Infatti, per ogni attributo di ogni tabella inserita nella query, nella pagina di composizione viene generata dalla servlet la riga del form di composizione dell’attributo in questione, con i suoi menù popolati dai valori salvati nel database. Questa riga viene caricata anche se tutti gli attributi della tabella a cui questo attributo appartiene non sono stati caricati con la relativa chiamata AJAX alla servlet GetTableFields (vedi sezione 6.2). Questo permette di risparmiare tempo non effettuando la chiamata AJAX per mostrare tutti gli attributi della tabella dell’attributo salvato nella query: infatti vengono solo caricati (ma non mostrati all’utente) i dati relativi a questo attributo. In questo modo questi dati, essendo presenti nel form di composizione, vengono inseriti nella query.

Inoltre, per ogni tabella coinvolta nella query, nell'interfaccia viene selezionato il checkbox relativo, e questo comporta un (seppur piccolo) ritardo nell'esecuzione del codice Javascript.
A titolo di esempio, si riportano alcuni tempi di esecuzione di "casi limite":

- feature **Transcript** con tutti gli attributi di tutte le tabelle selezionati:
  - tempo di esecuzione servlet: 261 millisecondi
  - tempo di esecuzione javascript: 364 millisecondi
  - tempo di esecuzione totale: 625 millisecondi
- feature **Biological Function Feature** con tutti gli attributi di tutte le tabelle selezionati:
  - tempo di esecuzione servlet: 1.902 millisecondi
  - tempo di esecuzione javascript: 1.125 millisecondi
  - tempo di esecuzione totale: 3.027 millisecondi
- features **Biological Function Feature e Gene** con tutti gli attributi di tutte le tabelle selezionati:
  - tempo di esecuzione servlet: 3.528 millisecondi
  - tempo di esecuzione Javascript: 4.333 millisecondi
  - tempo totale di esecuzione: 7.861 millisecondi

I tempi totali sono più alti rispetto al caricamento delle pagine di composizione di "default", ma rimangono comunque sotto i 10 secondi.

7.1.4 Tempi di generazione dell'interfaccia grafica

In questa sezione si riportano i tempi di esecuzione della pagina di composizione delle query in modalità "grafica", tramite grafo.
Come descritto nella sezione 6.3, molto di questa interfaccia è mutuato dalla modalità visuale i cui tempi di esecuzione sono stati riportati nella sezione precedente.
Un carico aggiuntivo è dato dal caricamento del grafo e dalla gestione dell'interazione con lo stesso.

Il caricamento della pagina visualSearch.jsp, che mostra il grafo pronto per la composizione della query, ha una durata di circa 300 millisecondi.
Come descritto nella sezione 6.3, ogni volta che l'utente clicca su un nodo o su un arco del grafo, viene mostrato il dialog di selezione degli attributi. Questo dialog è generato in modo analogo alla modalità "visuale" precedentemente descritta.

Inoltre, a ogni attivazione o disattivazione di un nodo o di un arco, viene modificato il grafo nascondendo o attivando i nodi interessati dall'azione svolta dall'utente.
Di conseguenza, i tempi di attivazione di un nodo saranno la somma dei tempi di generazione del suo dialog di selezione degli attributi con il tempo di modifica del grafo.

In Tabella 25 si riportano i tempi in millisecondi di attivazione di ogni nodo da una situazione "vergine" del grafo.

Feature	Tempo totale
Biological Function Feature	1.248
Clinical Synopsis	593
Dna Sequence	627
Enzyme	758
Gene	1.466
Gene Expression Feature	757
Genetic Disorder	843
Pathway	655
Protein	1.123
Protein Fam Dom	640
Small Molec	655
Transcript	655
Media	**835,00**
Deviazione standard	**286,17**

Tabella 25 - Modalità di composizione "grafica" delle query: tempi in millisecondi di attivazione di un nodo del grafo

Per quanto riguarda l'attivazione di più nodi, i tempi di esecuzione variano a seconda della situazione del grafo nel momento dell'attivazione: infatti, a seconda dei nodi e degli archi attivi, cambia anche il numero dei nodi e degli archi da attivare.

Caricamento di una query

Un caso interessante si ha quando si carica una query nell'interfaccia "grafica". In questo caso, i nodi (feature) e gli archi (associazioni) coinvolte nella query caricata vengono attivati tutti insieme, e vengono selezionati tutti gli attributi salvati nella query.
Si riporta il tempo di esecuzione della pagina visualSearch.jsp nel caso del caricamento di una query con le due features indicate nella sezione precedente:

- features **Biological Function Feature e Gene** con i soli attributi di default selezionati:
    - tempo totale di caricamento: 2.534 millisecondi

Caricando query che coinvolgono tre o più features, i tempi di caricamento della pagina tendono a salire:

- features **Biological Function Feature, Gene e Protein** con i soli attributi di default selezionati:
    - tempo totale di caricamento: 6.248 millisecondi
- features **Biological Function Feature, Gene, Protein e Genetic Disorder** con i soli attributi di default selezionati:
    - tempo totale di caricamento: 10.430 millisecondi

Come per la modalità "visuale", anche qui il numero di attributi e tabelle selezionate nella query influisce molto sui tempi di caricamento e generazione della pagina.

7.1.5 Discussione sui tempi di generazione delle interfacce di composizione della query

Nelle sezioni precedenti sono stati esposti i tempi di caricamento delle pagine per la composizione della query, sia in modalità "visuale" che in modalità "grafica".

Un dato interessante è stato rilevare come, per quanto riguarda la modalità "visuale", i tempi di caricamento della parte Javascript risultino più lunghi rispetto all'esecuzione della servlet che genera la pagina (vedi Tabella 23 e Tabella 24). Ciò è dovuto al fatto che la parte dell'algoritmo più onerosa è quella che si occupa di generare, tramite delle chiamate AJAX al server (alla servlet GetTableFields, vedi sezione 6.2), la lista degli attributi di ogni tabella. Questa scelta implementativa è stata fatta per velocizzare la composizione delle pagine di generazione della query con molti attributi selezionati, soprattutto in fase di caricamento di una query salvata in precedenza. Separando il caricamento della pagina dal caricamento singolo di ogni lista di attributi, è stato possibile frammentare il caricamento delle pagine. In questo modo, penalizzando lievemente i tempi di caricamento delle pagine di composizione di una singola feature (che rimangono comunque molto bassi, inferiori al secondo), è stato possibile ridurre i tempi di caricamento della pagina di composizione nel caso del caricamento di una query che coinvolgesse molte tabelle diverse.

Come descritto nelle sezioni precedenti, quindi, i tempi di caricamento delle pagine presuppongono un'attesa da parte dell'utente prima di poter comporre ed eseguire la query che intende sottoporre al GPDW. La domanda a cui è interessante fornire una risposta è la

seguente: è questa attesa accettabile, in riferimento al valore aggiunto dato dalla possibilità di comporre una query in modo visuale o grafico?

Per rispondere a questa domanda, è necessario innanzitutto chiedersi se i tempi di caricamento delle pagine sono accettabili oppure no.

Per determinare la validità dei tempi di risposta dati dall'applicazione si è deciso di adottare i criteri riportati in [24], che riportiamo in Tabella 26.

- **0.1 second** is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- **1.0 second** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- **10 seconds** is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

Tabella 26 - Limiti di tempo di risposta dell'applicazione [24]

Basandoci su questi criteri, possiamo definire i tempi di risposta ottenuti nelle sezioni precedenti:

- ***Modalità visuale***
  - **Singola feature**: la media dei tempi di caricamento delle pagine (vedi Tabella 23) di composizione delle query con una sola feature è di poco superiore al mezzo secondo (590 ms). Essendo inferiore al secondo, pur essendo percepita dall'utente, è un'attesa che non inficia la user experience degli utenti.
  - **Doppia feature**: in questo caso la media dei tempi di caricamento (vedi Tabella 24) è di due secondi. E' superiore alla soglia del secondo indicata precedentemente, ma è di molto inferiore rispetto alla soglia dei dieci secondi, dopo i quali l'attenzione dell'utente si sposta verso altre pagine. Può quindi essere considerata accettabile in questa applicazione.

- **Caricamento di una query**: la query più lenta ha un tempo di caricamento di circa otto secondi. E' un tempo considerevole, ma comunque inferiore a dieci secondi.

- ***Modalità grafica***
  - **Attivazione di una feature**: la media dei tempi di attivazione di una feature (vedi Tabella 25) è inferiore al secondo. Non influisce negativamente, quindi, sulla user experience degli utenti
  - **Caricamento di una query**: i tempi possono essere anche molto prolungati, e superiori ai dieci secondi, soprattutto in caso di query che coinvolgono molte feature. In questo caso l'esperienza d'uso degli utenti ne esce penalizzata. Tuttavia, la possibilità di vedere graficamente tutti gli elementi coinvolti nella query, e di poter eventualmente apportare alcune modifiche alla query prima di rieseguirla senza dover ricomporla dall'inizio, fornisce agli utenti un valore aggiunto molto grande alla loro navigazione nel GPDW, permettendo loro di intuire con un solo colpo d'occhio tutte le tipologie di dati offerti dal data warehouse e di progettare una query complessa in modo semplice e intuitivo.

Essendo i tempi di caricamento generalmente accettabili (anche se non sempre bassissimi), è lecito chiedersi se l'utente può accettare di aspettare qualche secondo pur di avere la possibilità di comporre una query in modo visuale (o grafico).
Sicuramente la possibilità di utilizzare un'interfaccia estende a molti utenti senza conoscenze del linguaggio di interrogazione SQL e della struttura del GPDW la possibilità di interrogare la grande mole di dati contenuta nel data warehouse. Inoltre l'interfaccia permette di tenere sotto controllo con maggiore chiarezza lo stato della query in composizione: feature selezionate, attributi inclusi, opzioni di esecuzione della query (tipo di join, distinct, ecc…). Anche la possibilità di salvare e caricare le query già composte è un aiuto in più agli utenti, che permette loro di non dover comporre di nuovo le query da zero, risparmiando così del tempo.
Riteniamo, quindi, che il trade-off temporale dato dal caricamento delle pagine di composizione della query sia minimo e sopportabile rispetto ai benefici che l'interfaccia offre agli utenti.

7.1.6 Tempi di processamento dei risultati

Questa applicazione, come descritto nella sezione 6.4.2, prevede un processamento dei risultati ottenuti, in modo da arricchire la loro visualizzazione di nuove possibilità.
Questo processamento richiede del tempo, in particolare in presenza di campi codificati (a cui va sostituito il loro valore non codificato) e campi a cui possono essere associati dei collegamenti ipertestuali (vedi sezione 6.4.2).
Per "tempo di processamento" si intende il tempo trascorso dalla fine dell'esecuzione della query generata dall'utente (cioè dal momento in cui sono disponibili i risultati) al momento in cui i risultati vengono visualizzati all'utente. E' espresso in millisecondi.

E' stata eseguita una query sulla singola feature "transcript", sulla sua tabella principale di feature con i soli attributi di default selezionati. In Tabella 27 vengono mostrati i tempi di processamento in diversi casi:

- caso con tutti gli attributi di default selezionati, che comprendono un campo a cui viene associato un collegamento ipertestuale (source_id), un campo codificato i cui dati sono salvati nella cache del server (source_name), un campo codificato i cui dati non sono salvati nella cache del server (taxonomy_id) e un campo non codificato (symbol)
- caso con un attributo con collegamento ipertestuale (source_id) e campo codificato con dati salvati nella cache del server (source_name)
- caso con attributo codificato i cui dati non sono nella cache del server (taxonomy_id) e campo non codificato (symbol)
- caso con solo campo non codificato (symbol)

Numero di righe visualizzate	Tutti gli attributi	Tutti gli attributi di default	Source_id e source_name	Symbol e taxonomy_id	Symbol
20	1.753	881	495	482	27
30	2.657	1.330	727	680	35
50	4.405	2.248	1.245	1.104	59
100	8.946	4.746	2.458	2.232	111
300	27.420	14.162	7.257	6.103	315
500	44.828	23.164	12.172	10.389	568
1000	85.700	41.892	24.300	20.795	1.106

Tabella 27 - Tempi di processamento dei dati (in millisecondi) estratti con una query sulla feature "transcript"

7.1.7 Discussione sui tempi di processamento dei risultati

Dai risultati riportati in Tabella 27 appare evidente come i tempi di processamento crescano all'aumentare del numero degli attributi selezionati, in particolare se questi attributi sono campi codificati oppure campi a cui possono essere associati dei collegamenti ipertestuali.

Fino a 100 righe visualizzate l'attesa può essere considerata accettabile, in quanto inferiore ai 10 secondi (vedi Tabella 26). Sopra le 100 righe visualizzate l'attesa comincia a essere maggiormente rilevante.

Si considera, in ogni caso, l'elaborazione effettuata necessaria per fornire all'utente dei dati comprensibili e utili per essere analizzati. Infatti, se si restituissero, ad esempio, dei dati con i soli valori codificati, l'utente capirebbe poco del loro significato.

Si ritiene, quindi, che l'attesa generata da questa elaborazione sia accettabile in termini di beneficio riportato dalla stessa. Inoltre, per una prima ispezione dei dati (20 righe visualizzate), l'elaborazione non richiede una attesa prolungata (inferiore ai due secondi).

7.1.8 Utilizzo della memoria del web server

Come illustrato nella sezione 5.1, l'applicazione web salva alcuni dati nella memoria del server, sia nello spazio della sessione sia nella cache del server, nel *context* dell'applicazione.

Monitorando lo stato dell'applicazione con PSI-Probe [25], uno strumento che permette di controllare la situazione del server Tomcat durante la sua esecuzione, si sono potuti evincere alcuni dati riguardo all'occupazione di memoria da parte dell'applicazione.

Occupazione della memoria da parte della sessione

Quando una sessione viene creata, con i suoi attributi di inizializzazione, ha una dimensione di 74kb. Nel momento di massima occupazione di memoria da parte della sessione, in fase di visualizzazione dei risultati (cioè quando, nel periodo di navigazione di un utente, nello spazio della sessione è salvato il maggior numero di attributi, vedi sezione 5.1.1), la sessione arriva a una dimensione massima di 175kb (Figura 41).

Quando un nuovo utente effettua il login nel sistema, viene creata una nuova sessione. Lo spazio occupato da questa nuova sessione va a sommarsi a quello delle sessioni già create.

Potenzialmente, quindi, lo spazio delle sessioni potrebbe crescere all'infinito. Però, essendo lo spazio occupato da ciascuna sessione molto piccolo, la capacità del server di ospitare contemporaneamente utenti è molto grande. Non dovrebbero sorgere quindi problemi di occupazione di memoria da parte delle sessioni.

Occupazione della memoria cache del server

Come spiegato nella sezione 5.1.2, alcuni dati vengono salvati nella cache del server, memorizzandoli nel *context* dell'applicazione. Questi dati sono disponibili a tutti gli utenti e sono i valori codificati di alcuni campi del GPDW.

Se non sono presenti nel context dell'applicazione, appena un utente ne fa richiesta (navigando nelle pagine di composizione della query) questi vengono aggiunti. Se invece sono già presenti, vengono semplicemente prelevati dal context.

Di conseguenza, questa occupazione di memoria ha un limite superiore che, a oggi, è di 134 attributi nel context (Figura 41). Questa occupazione di memoria, quindi, non può giungere a saturazione e non crea problemi.

Sessions	Ser.	Session attrs.	Session size	Context attrs.	Datasource usage
1	no	25	175 KB	134	0%

Figura 41 - Occupazione della memoria da parte dell'applicazione (screenshot da PSI-Probe [25])

7.2 Test di usabilità dell'applicazione web

Durante il lavoro di Tesi, per testare, validare e migliorare cio che è stato impementato, è stato progettato e realizzato un test di usabilità dell'interfaccia dell'applicazione. In questa sezione ne vengono illustrati la progettazione e i risultati.

7.2.1 Progettazione del test

In questa sezione vengono riportati i criteri attraverso i quali è stato progettato questo test.

Metriche utilizzate

Il test è stato pensato e progettato per valutare, qualitativamente, la navigabilità all'interno dell'applicazione, la facilità di comprensione e uso dell'interfaccia e le performance dell'applicazione stessa.
Quantitativamente, sono stati misurati:

- Il **Success Rate** dei task di questo test (0: fallimento, 0.5: successo parziale, 1: successo)
- Il **tempo impiegato** di ciascun utente per eseguire ogni task
- Il **tempo percepito** da ogni utente per eseguire ogni task
- Il numero di **errori** o "*undo*" di ogni utente per ciascun task

Metodo di raccolta dei dati

Il test è stato eseguito attribuendo a 9 utenti diversi 8 task da effettuare nell'applicazione, divisi in 3 scenari differenti. Gli utenti sono stati osservati durante l'esecuzione di questo task. A loro è stato richiesto di "pensare ad alta voce" (*thinking aloud*), per poter meglio interpretare ciò che stavano facendo e individuare eventuali punti critici dell'applicazione.
Alla fine del test, a ogni utente è stato richiesto di compilare un questionario riguardante la loro esperienza d'uso appena effettuata.

Compiti (task) assegnati

I task assegnati sono stati otto, divisi in tre scenari:

- **Scenario 1**: Si vuole fare un'indagine sul diabete mellito. Si vuole sapere quali sono i sintomi di questa patologia e salvare i dati ottenuti per future indagini.

  - **Task 1**: Trovare le informazioni disponibili nel database sulla patologia del diabete. Quanti record sono stati trovati?
  - **Task 2**: Trovare tutti i sintomi di questa patologia. Scriverne 3.
  - **Task 3**: Salvare la query costruita nel database, per un utilizzo futuro.

- **Scenario 2**: Due pazienti, uno affetto da distrofia muscolare e l'altro da sclerosi laterale amiotrofica (SLA), possono presentare sintomi simili. Nella sintomatologia delle due malattie, si vogliono individuare tratti fenotipici comuni.
  - **Task 1**: Trovare due sintomi in comune tra la SLA e la distrofia muscolare. La term position delle patologie cercate deve essere OTHER o LEAF.
  - **Task 2**: Caricare la query salvata nello scenario 1 e modificarla per capire se ci sono sintomi in comune tra SLA, distrofia muscolare e diabete mellito. Visualizzare anche il fenotipo di ciascun sintomo. Ci sono sintomi in comune? Se sì, quali?

- **Scenario 3**: Una donna ha ricevuto una diagnosi di cancro al seno. Dal sequenziamento del suo genoma, limitato ai geni BRCA1 e BRCA2, risulta avere una sola mutazione puntiforme (SNP) in BRCA2, che si sa essere associata ad un aumentato rischio di carcinoma alla prostata. Si vogliono individuare tutti i geni che si sanno coinvolti in entrambe le forme neoplastiche (cancro al seno e cancro alla prostata), i pathway in cui questi geni si sa che sono coinvolti e gli enzimi codificati da questi geni.
  - **Task 1**: Trovare tutti i geni che risultano coinvolti sia nel cancro al seno che nel cancro alla prostata.
  - **Task 2**: Per questi geni trovati, selezionarli in base al tipo "protein coding". Trovare i loro pathway e identificare tutti gli eventuali pathway comuni.
  - **Task 3**: Per tutti i geni trovati nel task 1, identificare la Gene Ontology in cui sono coinvolti e gli enzimi che sono associati a queste funzioni. Scrivere almeno una Gene Ontology con il suo tipo e il suo enzima associato.

Profili utente

I profili utente utilizzati in questo test possono essere divisi in tre categorie:

- Informatici con conoscenze di biologia e del GPDW: utenti con id **1**,**2**,**3**,**8**.
- Informatici senza conoscenze di biologia e del GPDW: utenti con id **4**,**6**,**7**,**9**.
- Biologi con conoscenza di applicazioni analoghe a quella oggetto di questo test. Utente con id **5**.

Tutte e tre queste categorie rappresentano potenziali utenti finali di questa applicazione:

- gli informatici possono essere chiamati a effettuare elaborazioni dei dati presenti nel GPDW: necessitano di conseguenza di uno strumento semplice da usare per effettuare una prima indagine nei dati effettivamente presenti nel database e per eventualmente validare i risultati da loro ottenuti
- i biologi possono utilizzare questa applicazione per i loro lavori di ricerca e divulgazione scientifica

7.2.2 Risultati quantitativi

Success rate

	Scenario 1			*Scenario 2*		*Scenario 3*		
	Task 1.1	*Task 1.2*	*Task 1.3*	*Task 2.1*	*Task 2.2*	*Task 3.1*	*Task 3.2*	*Task 3.3*
1	0,5	1	1	0,5	1	1	1	0,5
2	0,5	1	1	0,5	1	1	0,5	0,5
3	1	0,5	1	0,5	1	1	0,5	0,5
8	1	0,5	1	1	0,5	0,5	1	0,5
Profile average	0,75	0,75	1	0,63	0,88	0,88	0,75	0,5
4	0,5	0,5	1	0,5	0	1	1	0,5
6	1	1	1	1	1	1	1	1
7	0,5	0,5	1	0,5	1	1	1	1
9	1	1	1	0,5	1	1	1	0,5
Profile average	0,75	0,75	1	0,63	0,75	1	1	0,75
5	0,5	0,5	1	1	1	0,5	1	1
Profile average	0,5	*0,5*	*1*	*1*	*1*	*0,5*	*1*	*1*
Total average	*0,72*	*0,72*	*1*	*0,67*	*0,84*	*0,89*	*0,89*	*0,67*

Tabella 28 - Success rate per task

Success rate: (44 + (27*0,5)) / 72 = **80%**

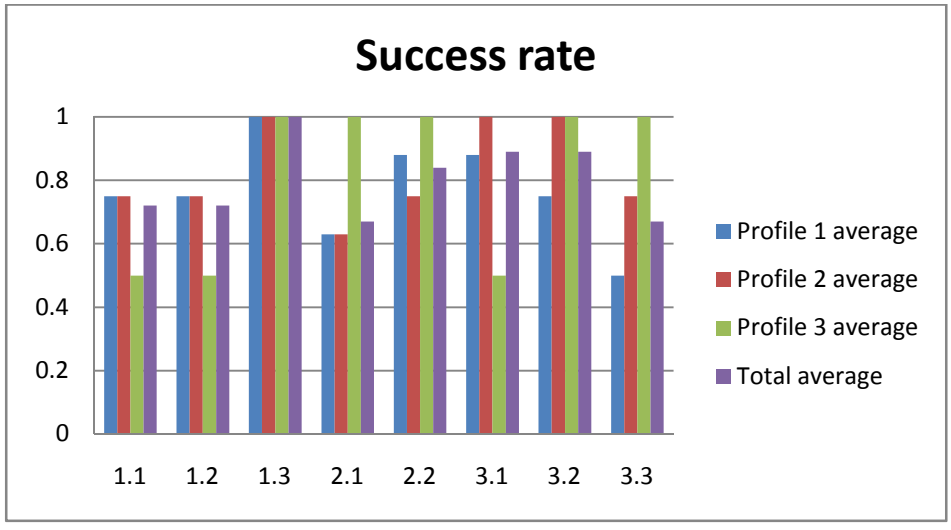


Grafico 1 - Success rate per task

Tempo per task (min)

	Scenario 1			*Scenario 2*		*Scenario 3*		
	Task 1.1	*Task 1.2*	*Task 1.3*	*Task 2.1*	*Task 2.2*	*Task 3.1*	*Task 3.2*	*Task 3.3*
1	4	2	1	6	2	6	6	13
2	2	2	1	8	4	5	8	6
3	3	4	1	5	5	6	9	5
8	2	2	1	8	5	5	7	8
Profile average	2,75	2,5	1	6,75	4	5,5	7,5	8
4	7	5	1	6	3	7	7	8
6	3	2	1	6	3	8	8	10
7	4	8	1	10	5	7	8	8
9	5	2	1	7	3	3	5	13
Profile average	4,75	4,25	1	7,25	3,5	6,25	7	9,75
5	4	7	1	4	4	4	5	4
Profile average	4	7	1	4	4	4	5	4
Total average	3,8	4,1	1	6,4	3,8	5,5	6,8	7,9

Tabella 29 - Tempo per task (min)

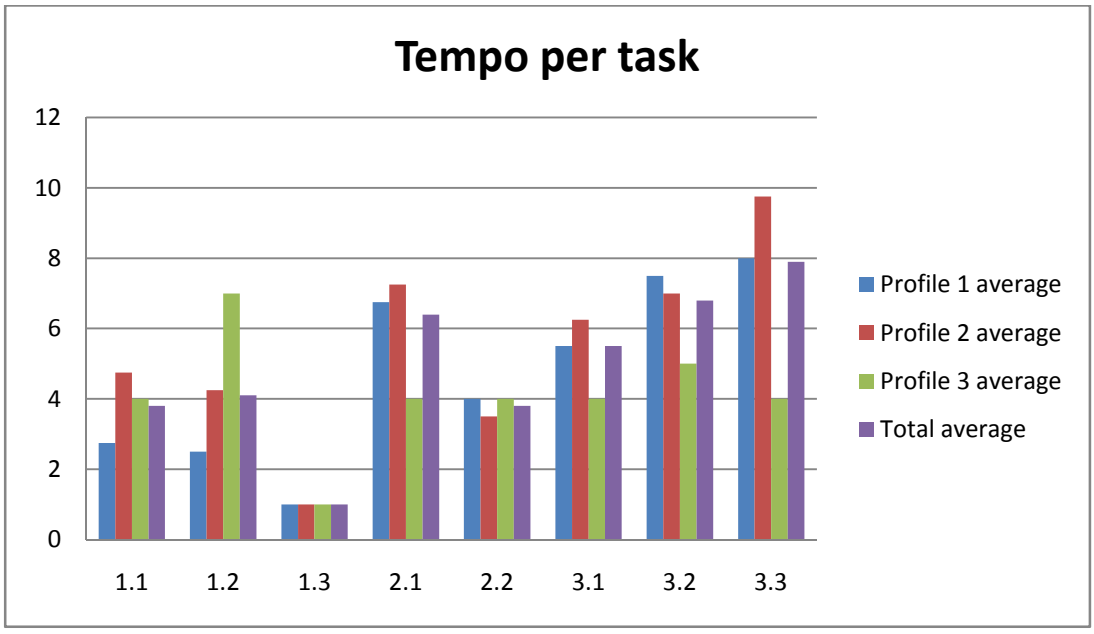


Grafico 2 - Tempo per task (min)

Errori o “undo” per task

	Scenario 1			*Scenario 2*		*Scenario 3*		
	Task 1.1	*Task 1.2*	*Task 1.3*	*Task 2.1*	*Task 2.2*	*Task 3.1*	*Task 3.2*	*Task 3.3*
1	1	0	0	1	0	1	0	2
2	1	0	0	1	0	0	0	1
3	0	1	0	0	0	0	0	2
8	0	1	0	1	2	0	0	3
Profile average	0,5	0,5	0	0,75	0,5	0,25	0	2
4	0	0	0	0	0	0	0	2
6	0	0	0	0	0	0	1	2
7	0	1	0	1	0	0	1	1
9	1	0	0	2	0	0	0	2
Profile average	0,25	0,25	0	0,75	0	0	0,5	1,75
5	1	0	0	0	0	0	0	0
Profile average	1	0	0	0	0	0	0	0
Total average	0,8	1	0,1	1	0,6	0,5	0,7	1,9

Tabella 30 - Errori o "undo" per task

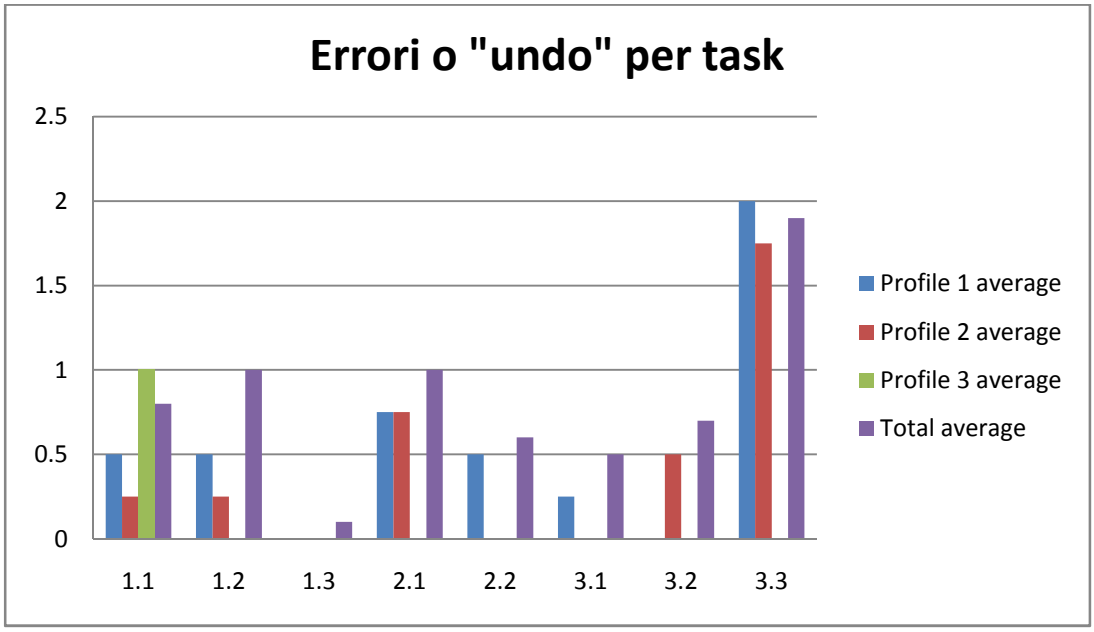


Grafico 3 - Errori o "undo" per task

Tempo percepito per task (1: molto, 5: poco)

	Scenario 1			*Scenario 2*		*Scenario 3*		
	Task 1.1	*Task 1.2*	*Task 1.3*	*Task 2.1*	*Task 2.2*	*Task 3.1*	*Task 3.2*	*Task 3.3*
1	4	2	2	5	1	2	2	3
2	2	2	1	4	2	3	2	3
3	4	4	1	3	2	3	3	3
8	3	4	2	4	3	4	4	4
Profile average	3,25	3	1,5	4	2	3	2,75	3,25
4	3	4	2	3	3	3	3	2
6	2	2	1	3	2	3	2	3
7	5	5	1	4	3	2	4	3
9	5	4	3	5	3	2	2	5
Profile average	3,75	3,75	1,75	3,75	2,75	2,5	2,75	3,25
5	5	3	1	2	3	3	3	4
Profile average	5	3	1	2	3	3	3	4
Total average	3,7	3,3	1,5	3,7	2,4	2,8	2,8	3,3

Tabella 31 - Tempo percepito per task

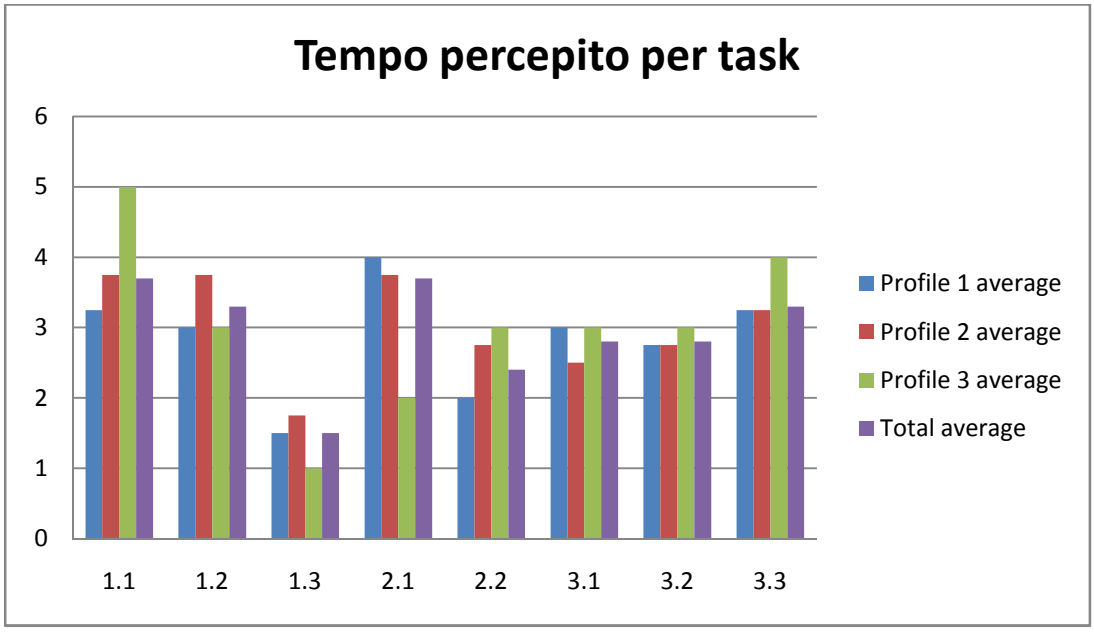


Grafico 4 - Tempo percepito per task

Questionario post-test

La valutazione di ogni domanda va da 1 (Non sono d'accordo) a 5 (Sono d'accordo).

Domande generali sull'esperienza d'uso:

	Utente:	***1***	***2***	***3***	***8***	***Avg.***	***4***	***6***	***7***	***9***	***Avg.***	***5***	***Avg.***	***Total avg.***
1	Sono soddisfatto dell'esperienza d'uso di questa applicazione	*4*	*4*	*4*	*4*	***4***	*4*	*3*	*4*	*4*	***3,75***	*5*	***5***	***4***
2	Penso che userei spesso questa applicazione	*4*	*3*	*3*	*4*	***3,5***	*4*	*3*	*5*	*1*	***3,25***	*5*	***5***	***3,56***
3	Questa applicazione è semplice da usare	*3*	*3*	*2*	*3*	***2,75***	*3*	*4*	*3*	*2*	***3***	*3*	***3***	***2,89***
4	La navigazione nell'applicazione è intuitiva	*3*	*4*	*2*	*2*	***2,75***	*4*	*3*	*3*	*3*	***3,25***	*3*	***3***	***3***
5	Non credo che avrei bisogno di assistenza per usare questa applicazione	*2*	*4*	*2*	*3*	***2,75***	*3*	*2*	*1*	*2*	***2***	*2*	***2***	***2,33***
6	Mi sono trovato bene nell'usare questa applicazione	*4*	*5*	*3*	*4*	***4***	*4*	*4*	*2*	*3*	***3,25***	*4*	***4***	***3,67***
7	Non ho dovuto imparare probabilmente troppe cose sull'uso di questa applicazione	*4*	*5*	*3*	*4*	***4***	*4*	*5*	*4*	*3*	***4***	*3*	***3***	***3,89***
8	La terminologia usata nell'interfaccia è comprensibile e coerente in tutta l'applicazione	*5*	*3*	*2*	*5*	***3,75***	*5*	*5*	*3*	*5*	***4,5***	*3*	***3***	***4***
9	Le pagine si caricano in modo sufficientemente rapido	*5*	*3*	*4*	*4*	***4***	*5*	*5*	*5*	*5*	***5***	*5*	***5***	***4,56***

Domande sulla composizione della query in modalità visuale:

	Utente:	***1***	***2***	***3***	***8***	***Avg.***	***4***	***6***	***7***	***9***	***Avg.***	***5***	***Avg.***	***Total avg.***
1	La composizione della query in modalità visuale è intuitiva	*3*	*5*	*3*	*4*	***3,75***	*5*	*3*	*3*	*3*	***3,50***	*3*	***3***	***3,56***
2	Penso che l'impostazione dei filtri su ogni campo sia facile da usare	*5*	*5*	*2*	*2*	***3,50***	*5*	*2*	*2*	*3*	***3,00***	*5*	***5***	***3,44***

Domande sulla composizione della query in modalità grafica:

	Utente:	***1***	***2***	***3***	***8***	***Avg.***	***4***	***6***	***7***	***9***	***Avg.***	***5***	***Avg.***	***Total avg.***
1	La modalità di ricerca su grafo è utile per comprendere le possibilità d'uso dell'applicazione	*5*	*5*	*4*	*5*	***4,75***	*4*	*5*	*5*	*5*	***4,75***	*5*	***5***	***4,78***
2	La modalità di ricerca su grafo è semplice da usare	*2*	*4*	*5*	*4*	***3,75***	*4*	*5*	*5*	*4*	***4,50***	*5*	***5***	***4,22***
3	La modalità di ricerca su grafo è intuitiva	*2*	*4*	*5*	*3*	***3,50***	*4*	*5*	*5*	*5*	***4,75***	*5*	***5***	***4,22***

Domande sulla visualizzazione dei dati:

	Utente:	***1***	***2***	***3***	***8***	***Avg.***	***4***	***6***	***7***	***9***	***Avg.***	***5***	***Avg.***	***Total avg.***
1	La modalità di visualizzazione dei dati aiuta a mettere in evidenza i risultati più interessanti	*4*	*3*	*3*	*3*	***3,25***	*5*	*3*	*3*	*4*	***3,75***	*4*	***4***	***3,56***
2	Il filtraggio dei risultati è semplice da usare	*4*	*5*	*2*	*3*	***3,50***	*5*	*3*	*2*	*4*	***3,50***	*4*	***4***	***3,56***
3	Il filtraggio dei risultati è utile per mettere in evidenza i dati interessanti	*4*	*5*	*2*	*4*	***3,75***	*5*	*4*	*2*	*5*	***4,00***	*4*	***4***	***3,89***

Domande sul salvataggio e caricamento delle query:

	Utente:	***1***	***2***	***3***	***8***	***Avg.***	***4***	***6***	***7***	***9***	***Avg.***	***5***	***Avg.***	***Total avg.***
1	La possibilità di usare query predefinite da altri è interessante	*5*	*4*	*4*	*5*	***4,50***	*4*	*5*	*3*	*5*	***4,25***	*5*	***5***	***4,44***
2	Il salvataggio e il caricamento delle query sono utili	*5*	*5*	*4*	*5*	***4,75***	*4*	*5*	*5*	*5*	***4,75***	*5*	***5***	***4,78***
3	Salvare a caricare una query è semplice e immediato	*5*	*5*	*4*	*5*	***4,75***	*5*	*5*	*5*	*5*	***5,00***	*5*	***5***	***4,89***

7.2.3 Risultati qualitativi

Osservando e intervistando gli utenti durante il test sono stati riscontrati alcuni punti importanti che meritano una discussione.
Li discuteremo dividendoli a seconda degli scenari del test.

Scenario 1

In questo scenario gli utenti si sono confrontati soprattutto con la modalità visuale di composizione della query e con il salvataggio della query composta.

Task 1.1

Nella pagina di composizione della query, lo strumento più delicato, in termini di usabilità, si è rivelato essere il campo di testo di tipo “auto complete”.
Tale campo è un campo che fornisce i suggerimenti per l’inserimento di valori nel filtro dell’attributo selezionato dall’utente. Ogni valore viene inserito nel campo come un “token” indivisibile, e l’utente può sceglierlo sia tra i suggerimenti proposti sia “forzare” l’inserimento di ciò che ha digitato (Figura 42 - Autocomplete a "token"Figura 42).

Figura 42 - Autocomplete a "token"

Non tutti gli utenti hanno compreso, da soli, come usare correttamente questo tipo di campo. La maggior parte di loro ha avuto bisogno di assistenza.
Inoltre, sempre per questo tipo di campo, è necessario cliccare sulla label “Show field” (Figura 43) per mostrare lo spazio in cui inserire i valori. Non tutti gli utenti hanno compreso immediatamente il significato di questa label.

Figura 43 - Label "show field"

Sempre per quanto riguarda l’autocomplete, in fondo al menu dei suggerimenti sono mostrate tre opzioni diverse dalle altre, definite “wild character” (Figura 44), che permettono la ricerca di tutti i termini che iniziano, finiscono, o contengono il termine digitato. Alcuni utenti non si sono accorti di questa possibilità, altri invece non hanno capito cosa significasse il suggerimento.

Figura 44 - Suggerimento termine "wild character"

Infine, e questo è un problema che ha riguardato in linea di massima di tutti gli utenti e tutti i task coinvolti, si è riscontrata una certa difficoltà da parte degli utenti di trovare il link attraverso cui "tornare indietro" alla pagina iniziale di selezione delle feature.

Molti utenti non hanno identificato la label "GPKB Data" (Figura 45) come possibilità di tornare all'inizio, e hanno usato il tasto "back" del browser.

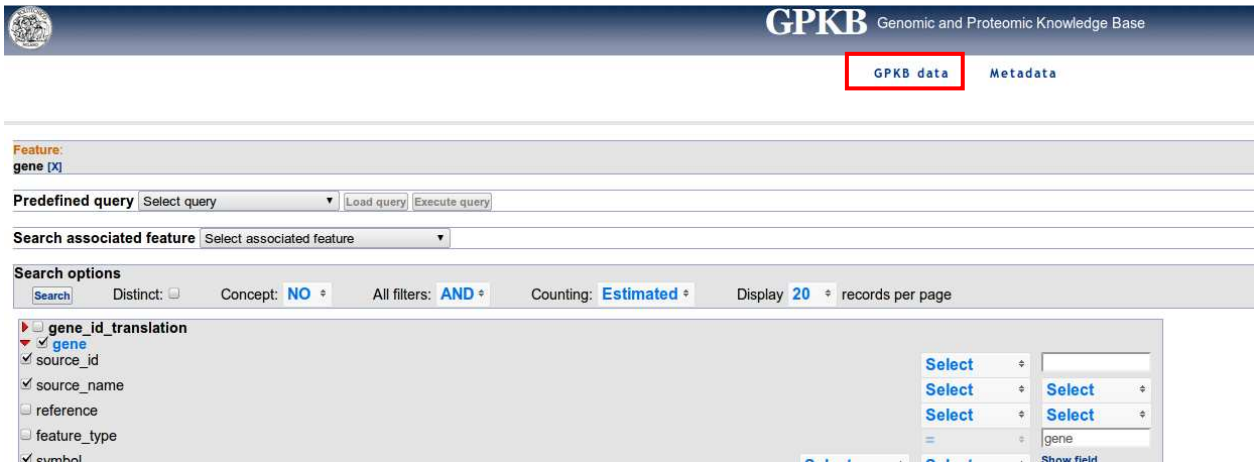


Figura 45 - Link per tornare alla pagina iniziale

Task 1.2

In questo task gli utenti hanno dovuto estendere la query da una a due feature.

Alcuni utenti, specialmente quelli senza conoscenze biologiche o che non conoscevano la struttura del database GPDW, non hanno capito il significato dei nomi delle features indicate (es: genetic_disorder significa "patologie").

Alcuni utenti non hanno esteso la query in fase di composizione, tramite l'apposito menu, ma hanno preferito ricominciare da capo.

Task 1.3

Durante questo task nessun utente ha presentato particolari difficoltà. Tutti sono riusciti a salvare la query in modo semplice e veloce.

Scenario 2

In questo scenario gli utenti hanno dovuto utilizzare anche e soprattutto la pagina di visualizzazione dei risultati, sfruttandone le diverse caratteristiche.

La situazione di partenza è quella finale dello scenario 1, quindi la schermata di visualizzazione dei risultati.

Task 2.1

Durante questo task, molti utenti hanno usato diverse modalità per tornare indietro alla pagina iniziale.

Qualcuno ha usato il tasto "back" del browser, qualcun altro il bottone "Back to query composition" (Figura 46) presente nella pagina di visualizzazione dei dati, qualcuno ha utilizzato le "x" in cima alla pagina di composizione della query (Figura 47) per eliminare entrambe feature dalla ricerca e tornare alla pagina iniziale.

Figura 46 - Bottone "back to query composition"

Figura 47 - Bottoni di rimozione feature

Dopo la composizione ed esecuzione della query, nella pagina di visualizzazione quasi tutti gli utenti hanno utilizzato la possibilità di ordinare i dati della tabella per identificare i sintomi in comune tra la due malattie indicate.

In due casi i filtri indicati sono stati inseriti non in fase di composizione bensì in fase di visualizzazione dei risultati.

Task 2.2

Nessuno degli utenti ha presentato difficoltà nel caricamento della query salvata nello scenario precedente.

Scenario 3

In questo scenario gli utenti si sono confrontati con una più complessa composizione della query, che ha richiesto anche l'utilizzo della modalità grafica di composizione delle query.

Task 3.1

Questo task è molto simile ai precedenti. Qualcuno, già in questa fase, ha preferito utilizzare la modalità grafica di composizione della query rispetto a quella visuale.

In fase di visualizzazione dei dati a qualche utente è sfuggito qualche risultato, e alcuni hanno lamentato un insufficiente aiuto da parte dell'applicazione per l'estrazione dei dati.

Task 3.2

Alcuni utenti già in questa fase hanno esteso la query tramite il grafo.

Qualcuno ha utilizzato i filtri in fase di visualizzazione, senza problemi.

Task 3.3

Per completare questo task, tutti gli utenti hanno dovuto comporre (o estendere) la query tramite la modalità grafica.

Grossomodo la metà degli utenti ha avuto difficoltà nel comprendere la label "Show query in visual feature search" (Figura 48), che permettere di mostrare (e quindi estendere) la query composta su una o due feature in modo visuale nella modalità grafica. Alcuni hanno preferito ricominciare dall'inizio e comporre interamente la query tramite il grafo.

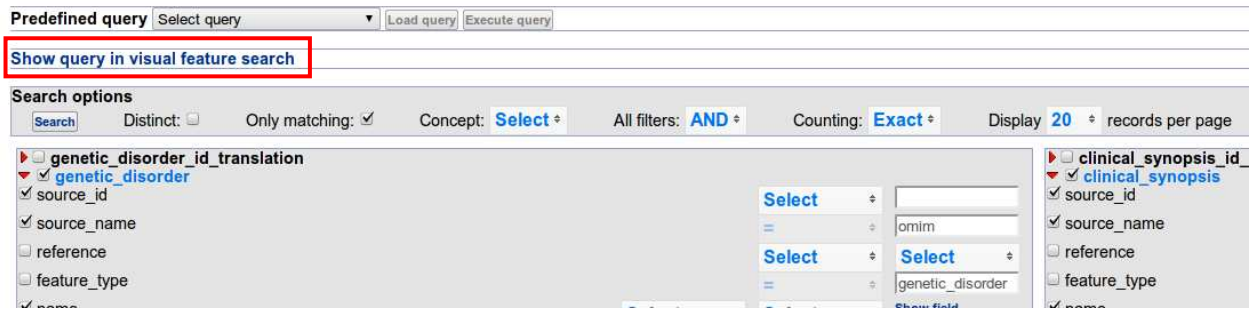


Figura 48 - Label "Show query in visual feature search"

L'interazione con il grafo da parte degli utenti è avvenuta senza particolari problemi. Tutti gli utenti si sono mostrati, a un primo impatto, "titubanti" e indecisi nel comprendere il significato di alcune icone.

Il problema principale, che tutti gli utenti hanno evidenziato, è stato nell'accorgersi dell'attivazione automatica di un arco del grafo (non necessario ai fini della query) durante le selezione di una feature. Questa attivazione comprometteva l'esito della query stessa, e tutti gli utenti si sono mostrati disorientati e hanno avuto bisogno di assistenza. Il problema è mostrato in Figura 49 e Figura 50.

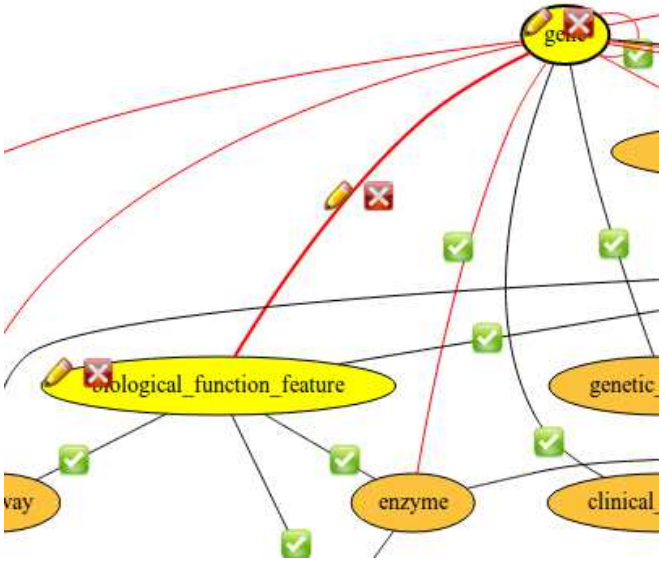


Figura 49 - Attivazione della feature enzyme....

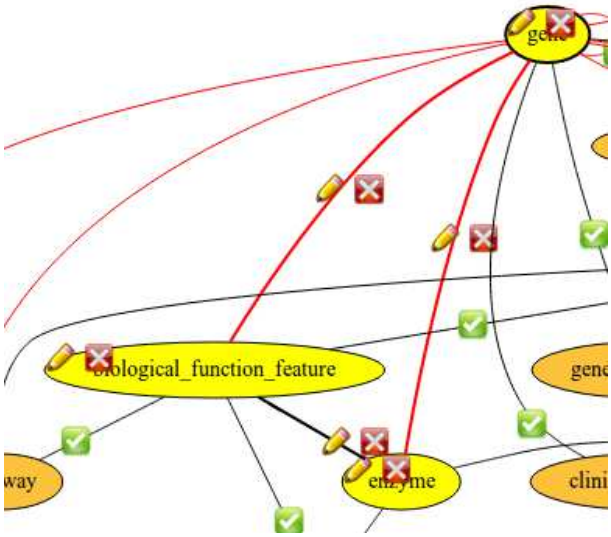


Figura 50 - ...vengono attivati automcaticamente entrambi gli archi, sia tra biological_function_feature e enzyme che tra gene e enzyme

7.2.4 Discussione dei risultati

I risultati riscontrati, sia quantitativi che qualitativi, hanno evidenziato alcuni problemi di usabilità comuni da parte di tutti gli utenti. Nessuno di questi, però, ha impedito agli utenti di portare a compimento con successo la maggior parte dei task assegnati.

Un alto **success rate**, dell'***80%***, infatti, evidenzia il fatto che i problemi che gli utenti hanno incontrato durante l'esecuzione del test non sono stati così critici da "bloccare" le loro attività. Solo in un caso, un task non è stato portato a termine. Non si è evidenziata una grande differenza tra i diversi profili utente sotto questo aspetto.

Il **tempo impiegato** per ogni task si è rivelato più alto della media nei task *2.1* e *3.3*. Il task 2.1 è il primo task che richiedeva la visualizzazione e l'analisi dei risultati, e gli utenti hanno mostrato qualche difficoltà nel capire ed evidenziare i dati a loro richiesti.

Il task 3.3 è il più complesso dell'intero test, e richiede l'utilizzo della modalità grafica di composizione della query. Molti utenti si sono mostrati "titubanti" davanti al grafo delle features, non capendo bene cosa potessero o dovessero fare.

In questo caso, si è notata la differenza tra i diversi profili utente. Gli utenti *informatici senza conoscenza del GPDW* hanno impiegato in media più tempo per completare i task rispetto agli altri profili. Questo perché hanno impiegato più tempo nel comprendere la terminologia e le potenzialità dell'interfaccia.

Il **numero di errori**, o "undo", medio non è stato alto, per nessun utente. La loro osservazione ha dimostrato che, seppur a volte dopo una lungo esame dell'interfaccia con cui avevano a che fare, nella maggior parte dei casi gli utenti hanno effettuato la scelta corretta.

Il **tempo percepito** dagli utenti per ogni task si è rivelato, mediamente, né troppo lungo né molto breve. Anche in questo caso molto è dipeso dall'esperienza maturata sull'uso della applicazione tramite lo svolgimento dei task precedenti.

Dal **questionario** somministrato agli utenti dopo la conclusione del test, si possono trarre alcune interessanti conclusioni:

- La maggior parte degli utenti è stata *soddisfatta dell'esperienza d'uso* di questa applicazione. La media delle valutazioni, infatti, è alta (4 su 5)
- Delle domande generali sull'esperienza d'uso, solo due hanno un valore medio sotto il 3 (punteggio medio). Gli utenti, infatti, lamentano una *eccessiva complessità dell'interfaccia* e affermano che *avrebbero bisogno di assistenza* nell'utilizzo di questa applicazione

- Un risultato interessante, dal punto delle metriche utilizzate, è la *velocità percepita di caricamento delle pagine*. Tutti gli utenti sono stati soddisfatti di questo aspetto: le media delle valutazioni infatti è superiore a 4,5.

Per quanto riguarda la **modalità di composizione visuale della query**, questa *non ha riscosso molto gradimento* da parte degli utenti, ottenendo un punteggio medio di 3,5.

Di contro, la **modalità grafica** ha ottenuto punteggi superiori a 4, mostrandosi *preferita* dagli utenti soprattutto per la globalità della rappresentazione dei dati in cui effettuare la ricerca.

Anche la **modalità di visualizzazione dei dati** non ha ricevuto punteggi molto alti, e probabilmente dovrà essere migliorata.

Infine, la possibilità di **salvare nel database e caricare le query in composizione** è stata recepita e utilizzata con successo dagli utenti, che la ritengono utile e semplice da usare. Ciò è testimoniato anche dai *bassi tempi di completamento* e dall'*alto success rate* dei task relativi a questa caratteristica dell'applicazione.

Qualitativamente, osservando gli utenti sono stati riscontrati alcuni problemi ricorrenti:

- **Difficoltà di ritorno alla pagina inziale**: nessuno degli utenti ha capito facilmente come tornare alla pagina di selezione delle feature, sia partendo dalla pagina di composizione della query che dalla pagina di visualizzazione dei risultati.
- **Difficoltà di espansione della query a due o più feature**: molti utenti non hanno capito la possibilità di espandere la query in composizione aggiungendo una o più feature. Le label "*Search associated feature*" e "*Show query in visual feature search*" non si sono dimostrate sufficientemente chiare.
- **Difficoltà di comprensione dei termini di ricerca**: alcuni utenti, soprattutto quelli senza conoscenze biologiche, si sono trovati spaesati davanti ai nomi dei campi delle tabelle, ignorandone il significato.
- **Difficoltà di utilizzo dei campi "auto complete"**: la maggior parte degli utenti si è trovata in difficoltà nell'utilizzo dei campi di tipo "auto complete". Molti sbagliavano l'inserimento dei dati, o non aspettavano il suggerimento dato dal sistema.
- **Macchinosità dell'estrazione di dati interessanti dalla pagina di visualizzazione**: molti utenti hanno lamentato una *scarsa possibilità di elaborazione dei dati* ottenuti. La sola visualizzazione tabellare si è rivelata insufficiente per una buona esperienza d'uso
- **Difficoltà di comprensione dei meccanismi di attivazione del grafo**: tutti gli utenti si sono mostrati *spaesati* davanti al grafo delle features. Tutti *non si sono accorti* dell'attivazione automatica degli archi di associazione tra tutte le features selezionate. Inoltre non tutti hanno capito immediatamente il significato e la funzione di tutte le

icone presenti sul grafo. Qualcuno di loro ha lamentato una *eccessiva presenza e ambiguità di queste icone*, alcune delle quali si sovrapponevano fra di loro

Osservando gli utenti, è stato interessante rilevare come ognuno di loro usasse un *modo diverso per comporre le query* utilizzando una delle diverse opzioni di navigazione offerte dall'applicazione: qualcuno estendeva la query già composta, altri cominciavano da zero, altri preferivano l'utilizzo del grafo rispetto alla modalità visuale. Queste differenze si sono mostrate attraverso tutti i profili utente.

7.2.5 Suggerimenti per il redesign

Tramite la discussione dei risultati sono stati evidenziati una serie di problemi per cui è necessario pensare un nuovo design.

Alcuni utenti, specialmente quelli senza conoscenze biologiche o della struttura del GPDW, non hanno capito immediatamente il significato di alcune label, della possibilità di associare due features, di filtrare i campi.

Può essere utile pensare e progettare una diversa interfaccia, di base, meno "potente" in termini di definizione della query ma più "user friendly", pensata per quegli utenti interessati all'utilizzo dell'applicazione ma senza competenze specifiche che invece sono importanti per utilizzare al meglio l'interfaccia attuale.

Per quanto riguarda i problemi dell'interfaccia attuale, si forniscono i seguenti suggerimenti:

- **Difficoltà di ritorno alla pagina inziale**: è necessario definire una label chiara e inequivocabile che permetta di tornare alla pagina di selezione delle feature. La label attuale, "GPKB Data", non è espressiva e comprensibile.
- **Difficoltà di espansione della query a due o più feature**: bisognerebbe evidenziare e spiegare questa possibilità all'utente. La label "Search assocated feature" con il menù associato non è sufficiente. Può essere utile inserire un piccolo paragrafo di spiegazione di questa possibilità e integrare la label con una icona intuitiva ed esplicativa.
- **Difficoltà di comprensione dei termini di ricerca**: può essere utile, quando possibile, fornire una spiegazione dei campi delle tabelle su cui si sta effettuando la ricerca. Le label che li contraddistinguono, spesso, non sono sufficientemente chiare.
- **Difficoltà di utilizzo dei campi "auto complete"**: è necessario ripensare e riprogettare questo tipo di campi, mantenendo il più possibile la libertà degli utenti di inserire i valori che loro vogliono, cercando di "interpretare" la loro volontà senza forzarli a effettuare macchinose selezioni per esprimere le loro intenzioni di ricerca. Può essere utile estendere lo spazio dedicato a questi campi, separando il menu dei suggerimenti da quello di inserimento dei valori, inserendo anche qualche icona esplicativa.
- La label "Show field" tramite la quale viene visualizzato il campo di ricerca non è sufficientemente chiara e in evidenza. E' necessario cambiarla e renderla più esplicita (visualizzando anche il numero degli eventuali valori inseriti).

- **Macchinosità dell'estrazione di dati interessanti dalla pagina di visualizzazione**: la visualizzazione tabellare si è dimostrata insufficiente per l'estrazione dei dati. Potrà essere interessante implementare nuove modalità di ricerca e evidenziazione dei dati, isolando automaticamente, per esempio, tutti i dati associati a un certo valore, o permettendo un migliore e più flessibile filtraggio della query.
- **Difficoltà di comprensione dei meccanismi di attivazione del grafo**: è importante fare capire agli utenti ciò che avviene durante l'interazione con il grafo, mostrando un avviso di "loading" durante le fasi di elaborazione dei dati e chiedendo all'utente il consenso prima di selezionare automaticamente eventuali archi o nodi.
- Può essere utile, per l'utente, affiancare al grafo un paragrafo di help, che spieghi brevemente con funziona l'interazione con il grafo e la selezione degli attributi, in modo che l'utente non si ritrovi completamente "spaesato" di fronte al solo grafo delle features.

In ogni caso, questo test ha evidenziato la necessità di un preliminare, seppur breve, addestramento degli utenti prima della sua esecuzione. Diventa quindi fondamentale generare una pagina di help e la possibilità di mostrare dei piccoli popup di aiuto "in itinere" a richiesta dall'utente, che mostrino la spiegazione delle diverse caratteristiche dell'applicazione.

8 Conclusioni

Questa Tesi ha portato alla realizzazione di un'applicazione web che permette di definire ed effettuare facilmente in modo visuale interrogazioni al data warehouse GPDW, in modo ottimizzato e performante. L'applicazione, inoltre, può adattarsi, in modo trasparente agli utenti, a ogni diversa versione del GPDW che, per definizione, è un data warehouse in evoluzione.

Per raggiungere questo obiettivo, prima di procedere alla progettazione e implementazione degli algoritmi e delle nuove interfacce, è stato necessario reingegnerizzare l'applicazione web preesistente, realizzando nuove strutture dati e suddividendo il livello "dati" dell'applicazione dal livello "applicativo". Questo processo di reingegnerizzazione ha permesso di estendere in modo coerente e strutturato l'applicazione stessa, generando del codice comprensibile e facilmente estendibile.

Il lavoro, poi, si è sviluppato nella progettazione di un algoritmo di generazione dinamica delle query che generasse delle interrogazioni in SQL ottimizzate e performanti che potessero fornire una risposta all'utente in tempi adeguati, nonostante l'elevata dimensione di molte delle tabelle del data warehouse e il numero di join spesso richiesto. L'algoritmo ideato è stato implementato con successo nell'applicazione web, ed è ora possibile comporre visualmente ed eseguire query, anche complesse, all'interno del GPDW.

Per permettere agli utenti di comporre delle query articolate, anche su più di due features, sono stati sviluppati e implementati due tipi di interfacce, interoperanti e complementari: una di tipo "visuale", che permette la composizione di query su un massimo di due features associate, e una di tipo "grafico" che, tramite un grafo dinamicamente generato dai metadati del data warehouse, permette la composizione di query che coinvolgono più di due features. In qualunque momento, se lo stato (numero di features selezionate) della query lo permette, è possibile passare da un tipo all'altro di interfaccia di composizione.

E' stata inoltre arricchita la pagina di visualizzazione dei risultati: tramite l'analisi dei dati ottenuti come risultati di un'interrogazione e diverse query ai metadati, sono stati inseriti dei collegamenti ipertestuali su alcuni campi dei risultati, in modo da permettere all'utente di aprire pagine esterne all'applicazione web che forniscono informazioni aggiuntive sui dati ottenuti.

Per fornire all'utente il corretto numero di tuple ottenute da una interrogazione, è stata realizzata una nuova modalità di conteggio, che non rallenta l'esecuzione della query principale, ma che fornisce comunque, a scelta dell'utente, un valore preciso delle tuple ottenute entro un tempo ragionevole (definito in un file di configurazione dagli amministratori), oppure fornisce immediatamente un valore approssimato se l'utente non è interessato al conteggio preciso.
Infine, è stato esteso il database già esistente di gestione degli utenti per permettere il salvataggio di query definite dagli utenti e il caricamento di query precedentemente salvate.
Le performance dell'applicazione web si sono rivelate accettabili, sia in termini di esecuzione della query che in termini di caricamento delle pagine web.
Per testare la validità dell'interfaccia implementata e dell'applicazione in generale è stato eseguito un test di usabilità dell'interfaccia dell'applicazione web. Sono stati osservati nove utenti eseguire diversi scenari sull'applicazione, ed è stato sottoposto loro un questionario di valutazione. Tutti gli utenti sono riusciti a completare i compiti assegnati e si sono rivelati soddisfatti della propria esperienza d'uso. Sono stati evidenziati alcuni piccoli problemi di usabilità, riportati in questa Tesi, che verranno affrontati nei futuri sviluppi di questa Tesi. Nessuno degli utenti ha lamentato problemi di performance dell'applicazione, sia come lentezza nell'esecuzione delle query, sia nel caricamento delle pagine web.

Questo lavoro di Tesi ha permesso di migliorare l'applicazione web preesistente, fornendo la possibilità agli utenti di comporre ed eseguire query multifeature che forniscano risposte in tempi adeguati e accettabil per gli utenti stesso.
Gli algoritmi sviluppati sono flessibili, in modo tale che possano adattarsi a qualunque istanza del GPDW generata. In questo modo l'applicazione web non deve cambiare in seguito a cambiamenti nella base di dati.
L'interfaccia generata si è dimostrata fruibile e sufficientemente intuitiva per utenti con una certa familiarità con questo tipo di applicazione o comunque con l'utilizzo di applicazioni web interattive e dinamiche.

In conclusione, si considerano raggiunti gli obiettivi prefissi per questo lavoro di Tesi.
L'applicazione è stata arricchita di nuove funzionalità ed è diventata utilizzabile da parte degli utenti, per generare ed eseguire interrogazioni complesse al GPDW in tempi adeguati.
Questa applicazione è completamente funzionante e già disponibile al pubblico, presso l'url http://www.bioinformatics.polimi.it/GPKB/.

Si confida che possa risultare uno strumento utile per la ricerca e lo sviluppo di progetti attinenti al campo della bioinformatica e della biologia.

9 Sviluppi futuri

L'applicazione web realizzata in questo lavoro di Tesi ha migliorato ed esteso una versione già esistente della stessa. Molto è stato fatto, rendendo l'applicazione usabile e permettendo l'interrogazione di tutto il GPDW. Esistono ancora, però, molti margini di sviluppo: la mole di dati presente nel GPDW, infatti, è considerevole, ed è possibile permettere diverse esperienze di navigazione al suo interno.

L'interfaccia realizzata ed esposta agli utenti per comporre le query può essere definita "avanzata", in quanto richiede almeno limitate conoscenze bioinformatiche di dominio e una certa dimestichezza con applicativi simili per poter essere sfruttata nel pieno della sua funzionalità.
Un sicuro sviluppo, in futuro, sarà quello di progettare e realizzare un'interfaccia di "base", forse meno potente dell'attuale, ma più immediatamente comprensibile e utilizzabile da parte di utenti meno esperti.
Dai test realizzati è risultato che, nell'interfaccia di composizione delle query, l'utilizzo del campo "autocomplete" non è stato pienamente compreso dagli utenti. Sarà necessario progettare un campo simile che permetta, in modo semplice e intuitivo, l'inserimento di valori per la ricerca nel database, fornendo suggerimenti per l'inserimento (come già avviene), ma permettendo anche di inserire valori decisi dall'utente e, eventualmente, di poter inserire una lista di valori definiti precedentemente o prelevati da sorgenti esterne (copia/incolla, importazione di liste…).
La modalità di composizione grafica della query, tramite grafo, si è rivelata, da parte di alcuni utenti, un po' "ostica" e "macchinosa". Risulta, inoltre, ancora un po' lenta per quanto riguarda i tempi di interazione con il grafo. In futuro sarà necessario modificare o eventualmente riprogettare questa interfaccia per renderla più snella e immediatamente comprensibile agli utenti stessi.
La modalità di visualizzazione dei risultati è stata arricchita rispetto all'applicazione web di partenza, ma si può fare ancora molto per permettere agli utenti di gestire i risultati ottenuti in modo da mettere automaticamente in evidenza i dati più interessanti. Questo aspetto, che non ha riguardato lo sviluppo di questa Tesi, è tutto da studiare, progettare e implementare.

E' necessario, inoltre, creare una pagina di gestione delle query predefinite: al momento queste query sono inseribili soltanto dall'amministratore direttamente nel database.

Si ricordano, infine, tutti i suggerimenti per il redesign scaturiti dal test di usabilità dell'applicazione. In breve, si riportano qui di seguito:

- **Difficoltà di ritorno alla pagina inziale**: definire una label chiara e inequivocabile che permetta di tornare alla pagina di selezione delle feature.
- **Difficoltà di espansione della query a due o più feature**: bisognerebbe evidenziare e spiegare questa possibilità all'utente.
- **Difficoltà di comprensione dei termini di ricerca**: può essere utile, quando possibile, fornire una spiegazione dei campi delle tabelle su cui si sta effettuando la ricerca.
- **Difficoltà di comprensione dei meccanismi di attivazione del grafo**: importante fare capire agli utenti ciò che avviene durante l'interazione con il grafo, mostrando un avviso di "loading" durante le fasi di elaborazione dei dati e chiedendo all'utente il consenso prima di selezionare automaticamente eventuali archi o nodi.
- Affiancare al grafo un paragrafo di help, che spieghi brevemente con funziona l'interazione con il grafo e la selezione degli attributi.

10 Bibliografia

[1] Canakoglu A., Ghisalberti G., Masseroli M. Integration of Biomolecular Interaction Data in a Genomic and Proteomic Data Warehouse to Support Biomedical Knowledge Discovery Dipartimento di Elettronica e Informazione, Politecnico di Milano 2012

[2] Masseroli M, Ceri S, Tettamanti L, Campi A. *Model-driven modular integration of genomic and proteomic information*. IEEE Transaction on Knowledge and Data Engineering

[3] Masseroli M., Galati O., Pinciroli F. GFINDER: genetic disease and phenotype location statistical analysis and mining of dynamically annotated gene lists. 2005; p. 1-10

[4] *About PostgreSQL*. [Online]. http://www.postgresql.org/about/

[5] Gangi P. Sviluppo di un'applicazione web dinamica per la ricerca di informazioni in un grande datawarehouse bioinformatico Politecnico di Milano; 2011.

[6] Hall M. *More servlets and Javaserver pages*. Upper Saddle River, NJ: Prentice Hall; 2002.

[7] *Apache Tomcat*. [Online]. 1999 Disponibile su: URL: http://tomcat.apache.org/

[8] *About the Eclipse Foundation*. [Online]. Disponibile su: URL: http://www.eclipse.org/org/

[9] *PostgreSQL JDBC Driver*. [Online]. 2004; Disponibile su: URL: http://jdbc.postgresql.org/

[10] *JDOM*. [Online]. Disponibile su: URL: http://www.jdom.org/

[11] *Introducing JSON*. [Online]. Disponibile su: URL: http://www.json.org/

[12] *Apache Logging service project*. [Online]. Disponibile su: URL: http://logging.apache.org/log4j/

[13] *pgAdmin: PostgreSQL administration and management tools*. [Online]. Disponibile su: URL: http://www.pgadmin.org/

[14] Horstmann C, Cornell G. Core *Java 2: Volume I – Fondamenti.* 7th ed. Milano: Pearson Education Italia; 2005. p. 6, 98.

[15] Bergsten H. *Java Server Pages*. 3rd ed. Milano: Tecniche Nuove; 2004.

[16] Diotialevi F. *Java Enterprise Edition 5: Progetto e sviluppo di applicazioni web*. Milano:

Hoepli Informatica; 2006.

[17] *jQuery: the write less, do more, JavaScript Library*. [Online]. Disponibile su: URL: http://jquery.com/

[18] *DataTables (table plug-in for jQuery)*. [Online]. Disponibile su: URL: http://www.datatables.net/

[19] jQueryUI: a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. [Online] Disponibile su: URL: http://jqueryui.com

[20] *jQuery Tokeninput: a jQuery Tokenizing Autocomplete Text Entry*. [Online] Disponibile su: URL: http://loopj.com/jquery-tokeninput/

[21] *SVG: a widely-deployed royalty-free graphics format.* [Online] Disponibile su: URL: http://www.w3.org/Graphics/SVG/

[22] Ellison J., Gansner E., Koutsofios L., North S.C., Woodhull G. *Graph Drawing* Springer Berlin Heidelberg 2002; p. 483 - 484. Disponibile su: http://dx.doi.org/10.1007/3-540-45848-4_57

[23] *Graphviz: an open source graph visualization software*. [Online] Disponibile su: URL: http://www.graphviz.org/

[24] Nielsen J., *Usability Engineering* ed. Morgan Kaufman, San Francisco 1993. Cap. 5

[25] PSI-Probe , advanced manager and monitor for Apache Tomcat. [Online] Disponibile su: URL: http://code.google.com/p/psi-probe/