CAPITOLO 2 - Dai DBMS agli ORDBMS: l'impiego di PostgreSQL per realizzare l'applicazione.

Nell'introduzione abbiamo accennato brevemente a due tipologie di database: quelli relazionali e quelli ad oggetti. Dopo alcune considerazioni, siamo giunti alla conclusione che, per realizzare l'applicazione, nessuna delle due tipologie è quella più adatta allo scopo, poiché, sia le caratteristiche dell'una che dell'altra non sono sufficienti per implementare le funzionalità che l'applicazione deve fornire. Abbiamo quindi affermato che la soluzione ideale per costruire l'infrastruttura dell'applicazione è una tipologia di database che di fatto è un mix delle due, ovvero un database di tipo object-relational.

Questa tipologia di database può essere considerata come l'ultima evoluzione nel campo dei sistemi per la gestione dei dati, poiché, di fatto, cerca di sfruttare al massimo le potenzialità dei due modelli alla sua base, ed è per ciò che, questa nuova tipologia, per certi versi, può essere considerata una tipologia ibrida.

Ogni distinta tipologia di database, infatti, si basa concettualmente su un certo modello dei dati ben definito, mentre gli object-relational risultano essere una combinazione tra il modello relazionale ed il modello ad oggetti. Per comprendere al meglio le potenzialità offerte da questa tipologia di database, è senza dubbio utile conoscere i motivi che hanno portato alla loro nascita, cioè quali siano state le necessità contingenti che hanno determinato l'esigenza di dover creare questo nuovo tipo di database.

A tal proposito facciamo quindi un piccolo passo indietro, cerchiamo, infatti, di vedere velocemente come si è svolta l'evoluzione dei sistemi di gestione di basi di dati fino ad oggi.

2.1 - Tipologie ed evoluzione dei DBMS.

Per prima cosa è necessario introdurre il concetto di modello dei dati, al quale, tra l'altro, abbiamo già fatto riferimento poche righe sopra.

- Un **modello dei dati** è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che essa risulti comprensibile ad un elaboratore.

Il modello dei dati è quindi la base sulla quale implementare le caratteristiche di un qualsiasi DBMS o, più precisamente, possiamo dire che ogni modello dei dati fornisce dei meccanismi di strutturazione che permettono di definire nuovi tipi, attraverso l'utilizzo di tipi elementari predefiniti e costruttori di tipo.
Sempre seguendo il filo conduttore che ci lega all'obbiettivo di realizzare un'applicazione che permetta l'archiviazione e la ricerca di dati di tipo strutturato, dovrebbe risultare abbastanza chiaro che, lo scopo di questa trattazione teorica è quello di mettere in evidenza come un particolare modello di dati possa permettere o meno ad un sistema di gestione di basi di dati di gestire dati di tipo semi-strutturato e multimediale, cioè non solo tipi elementari. I principali modelli dei dati sono quattro ed ognuno ha una natura ben distinta che permette di caratterizzare in modo univoco ogni DBMS che vi appartiene:

- Il **modello gerarchico**, basato sull'uso di strutture ad albero (e quindi gerarchiche, da cui il nome), definito durante la prima fase di sviluppo dei DBMS (anni Sessanta), ma tuttora ampiamente utilizzato.
- Il **modello reticolare** (detto anche modello CODASYL, dal comitato di standardizzazione che lo definì con precisione), basato sull'uso di grafi, sviluppato successivamente al modello gerarchico (inizio anni Settanta).
- Il **modello relazionale** dei dati, attualmente il più diffuso, permette di definire tipi per mezzo del costruttore relazione, che consente di organizzare i dati in insiemi di record a struttura fissa.
- Il **modello a oggetti**, sviluppato negli anni Ottanta come evoluzione del modello relazionale, che estende alle basi di dati il paradigma di programmazione a oggetti.

Quasi tutti i DBMS in commercio utilizzano uno di questi modelli dei dati, che vengono anche detti modelli logici, poiché, come è stato appena detto, le strutture utilizzate da questi modelli, pur essendo astratte, riflettono una particolare organizzazione: ad alberi, a grafi, a tabelle o a oggetti.

Per l'utente finale di un'applicazione, basata sull'utilizzo di un database, non è comunque necessario conoscere il modello dei dati utilizzato per rappresentare e strutturare i medesimi nel database: l'utente finale interagisce con le funzionalità offerte dal DBMS mediante un'applicazione che, generalmente, presenta un'interfaccia, la quale rende completamente nascosto all'utilizzatore ciò che si trova sotto di essa. Per rendere meglio questo concetto è utile analizzare il modello stratificato riportato in figura 2.1: gli utenti non sono interessati a conoscere la rappresentazione logica o il tipo di memorizzazione fisica utilizzata dal DBMS per gestire i dati, ma solo ad utilizzare le funzionalità messe a disposizione dall'applicazione che si interfaccia al database. Se il DBMS riesce a mettere a disposizione le funzionalità necessarie affinché l'applicazione possa assolvere ai compiti per la quale è stata progettata, l'utente finale non ha il minimo interesse a conoscere con quale modalità il DBMS permetta l'esecuzione di certe funzionalità.

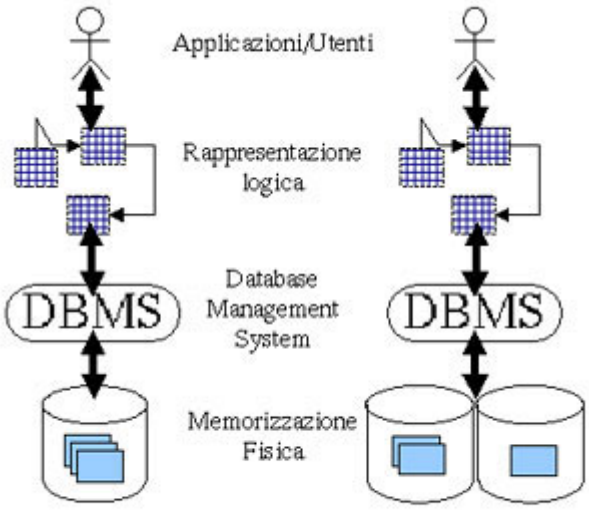

Figura 2.1 - La stratificazione dei DBMS

I problemi nascono quando, per nuove esigenze di utilizzo, si vogliono aumentare le funzionalità dell'applicazione e l'infrastruttura sottostante ad essa non lo permette. Ripensiamo all'esempio del commesso nel negozio di moda fatto nell'introduzione: se divenissero frequenti le richieste tipo quella fatta dal cliente nella circostanza descritta, cioè di vedere un'insieme di abiti partendo dall'immagine di un abito campione scelta come riferimento, sarebbe auspicabile che la direzione del negozio si adoperasse per inserire la funzionalità nel catalogo on-line, al fine di migliorare la qualità e l'efficienza dei servizi offerti nei propri negozi. Se però l'applicazione "catalogo on-line" si interfaccia ad un database basato su un modello dei dati che non permette l'implementazione di una simile funzionalità è un problema. Le soluzioni possibili a questa nuova necessità sono sostanzialmente due, ma, nelle condizioni operative ipotizzate, entrambe presentano comunque delle forti controindicazioni. La prima soluzione prevede di sostituire completamente l'applicazione con una nuova, che sia dotata da subito della funzionalità aggiuntiva richiesta; questo, però, comporta per l'azienda un investimento economico extra non preventivato, nonché dei problemi logistici dovuti alla migrazione dei dati dal vecchio al nuovo "catalogo on-line". La seconda consiste nel non rendere disponibile la funzionalità richiesta, poiché l'infrastruttura sulla quale poggia l'applicazione "catalogo on-line" non ne consente l'aggiunta. Questo inevitabilmente renderà il negozio impreparato a recepire i nuovi scenari di mercato e ciò costituisce sicuramente una limitazione ai servizi offerti dall'azienda, ponendo così un freno al suo sviluppo. Da questa ipotetica situazione si può quindi evincere che l'adozione di applicativi basati su infrastrutture che non consento l'estensione delle proprie funzionalità è una scelta che generalmente ha quasi sempre delle ripercussioni negative.

Detto ciò, è giusto sottolineare che i primi due tipi di database, quelli che utilizzano il modello gerarchico o reticolare, benché siano tuttora presenti in svariati applicativi, di fatto ormai concettualmente appartengono alla storia dell'informatica. La maggior parte dei database attualmente utilizzati oggi appartiene alla categoria dei database relazionali, per i quali sono già state accennate le potenzialità in sede di

introduzione. Anche quest'ultimi però, benché siano molto più versatili e performanti rispetto ai primi due, non offrono la possibilità di estendere le proprie funzionalità per la manipolazione di un qualsiasi nuovo tipo di dato. Cerchiamo quindi di capire fino a che punto può essere sufficiente l'utilizzo di un database relazionale e quando diventa invece necessario ricorrere ad un'altra tipologia.

Con l'avvento del modello relazionale e dei sistemi di basi di dati fondati su tale modello, fu fatto un passo molto importante per l'evoluzione di questi sistemi, in particolare, il modello relazionale permise di superare una limitazione comune sia al modello gerarchico, sia al modello reticolare: entrambi non permettevano di realizzare efficacemente la proprietà di indipendenza dei dati, proprietà di fondamentale importanza per realizzare applicazioni facilmente portabili e dotate di funzionalità strutturalmente sempre più complesse. Il modello relazionale, in virtù di questa caratteristica, è anche detto “modello basato su valori”. Questo significa che i riferimenti fra i dati in relazioni diverse sono rappresentati per mezzo di valori dei domini che compaiono nelle tabelle che caratterizzano le relazioni stesse. Differentemente, nel modello gerarchico e reticolare, le corrispondenze fra i dati sono invece realizzate mediante riferimenti espliciti fatti attraverso dei puntatori presenti nei record che caratterizzano i dati contenuti nel database.

Possiamo quindi riassumere dicendo che i database relazionali (RDBMS) hanno permesso la realizzazione efficace ed efficiente di applicazioni di tipo gestionale, caratterizzate principalmente da requisiti importanti quali la persistenza, la condivisione e l'affidabilità dei dati. Altre caratteristiche di rilievo per questo tipo di database sono sicuramente rappresentate dalla possibilità di implementare interrogazioni complesse, espresse mediante linguaggi dichiarativi come il SQL, e dalla possibilità di effettuare delle transazioni concorrenti sulla stessa base di dati. La necessità di un'ulteriore evoluzione, come abbiamo avuto modo di apprendere, nasce dal fatto che i RDBMS possono manipolare solo dati a struttura semplice, di tipo numerico/simbolico, e la rapida evoluzione tecnologica ha fatto emergere nuove esigenze applicative, per le quali la tecnologia relazionale è inadeguata, poiché, come è già stato detto a più riprese, la forte connotazione multimediale che

molte applicazioni stanno assumendo richiede a gran voce la possibilità di gestire dati di tipo strutturato. Questo scenario ha così suggerito l'introduzione nel mondo delle basi di dati di nozioni provenienti dal paradigma della programmazione orientata agli oggetti: da questa operazione sono nati i sistemi di gestione di basi di dati fondati sull'utilizzo del modello dei dati ad oggetti.

La prima generazione di ODBMS (Object Data Base Management System) è composta dai linguaggi di programmazione a oggetti persistenti, che realizzano solo alcune caratteristiche delle basi di dati, senza supporto per l'interrogazione, in modo quindi assai incompatibile con gli RDBMS. Gli ODBMS della seconda generazione realizzano invece un maggior numero di caratteristiche proprie delle basi di dati, e generalmente forniscono un supporto all'interrogazione. Possiamo quindi schematizzare i database fondati sul modello ad oggetti come divisi in due tecnologie di ODBMS: gli OODBMS (Object-Oriented), una tecnologia rivoluzionaria rispetto a quella degli RDBMS, e gli ORDBMS (Object-Relational), una tecnologia di evoluzione rispetto a quella degli RDBMS.

Considerando quanto detto fino ad ora, possiamo fare un'ulteriore passo in avanti affermando che una base di dati a oggetti è a tutti gli effetti una collezione di oggetti, dove ciascun oggetto ha un identificatore, uno stato, e un comportamento. L'identificatore OID (Object Identifier) garantisce l'individuazione in modo univoco dell'oggetto, e permette di realizzare dei riferimenti tra gli oggetti stessi. Lo stato è l'insieme dei valori assunti dalle proprietà dell'oggetto ed è in generale un valore a struttura complessa. Il comportamento, infine, è descritto dall'insieme dei metodi che possono essere applicati all'oggetto.

In questo contesto esula dai nostri interessi entrare ad analizzare in profondità con quali modalità opera un ODBMS, ma è senza dubbio importante evidenziare la caratteristica di questi sistemi di gestione dei dati per noi più importante: la possibilità di estendere i tipi di dati e di definire delle funzioni che operano su questi tipi. L'approccio object-oriented consente infatti l'uso di tipi e valori complessi e permette di associare ad un singolo oggetto una struttura qualunque. Viceversa, nel modello relazionale, alcuni concetti devono essere rappresentati obbligatoriamente

tramite più relazioni. Il modello relazionale puro richiederebbe inesorabilmente l'utilizzo di più relazioni per la rappresentazione di un dato strutturato e questa è una chiara limitazione del modello quando la base di dati è chiamata a manipolare molti dati strutturati. Nei modelli ad oggetti invece la situazione è facilmente gestibile, poiché un oggetto è una coppia (OID,Valore), dove l'OID è un valore atomico definito dal sistema e trasparente all'utente, mentre con "Valore" si può indicare un qualsiasi dato strutturato. Il valore assunto da una proprietà di un oggetto può essere quindi l'OID di un altro oggetto, realizzando così un riferimento anche tra oggetti, che possono anche derivare da istanze di classi diverse, al pari di ciò che avviene nel modello relazionale, quando si collegano tuple istanziate da tabelle che rappresentano relazioni diverse.

Tutto questo non deve far pensare che il modello ad oggetti abbia soppiantato il modello relazionale, poiché il modello relazionale, come abbiamo avuto modo di evidenziare, resta sempre quello più utilizzato, poiché le sue caratteristiche offrono tutt'oggi ampie garanzie di successo nella progettazione e nella realizzazione dei database. In particolare, la realizzazione di sistemi gestionali e contabili prevede quasi sempre l'utilizzo di un database relazionale ed in alcuni casi addirittura si riscontra tutt'oggi in tali settori la presenza di database gerarchici e reticolari. Questo è un chiaro indice del fatto che esistono ancora dei campi di applicazione che non necessitano di manipolare nuovi tipi di dati e, in mancanza di questa necessità, non emerge mai la volontà di un'ente o di un'azienda di sostituire il proprio sistema. Per quanto riguarda il modello ad oggetti lo possiamo invece considerare come un qualcosa in grado di sopperire alle mancanze del modello relazionale, mancanze che si manifestano in situazioni tipo quella nella quale ci troviamo adesso: gestire ed utilizzare dati di tipo strutturato.

Nel complesso quindi possiamo individuare nella tecnologia ORDBMS quella più adatta ai nostri scopi. Come abbiamo già detto, un database di tipo ORDBMS mantiene tutta la potenza del modello relazionale e permette l'introduzione e la manipolazione di nuovi tipi di dati, caratteristica che rappresenta sicuramente un punto di forza del modello ad oggetti.

Non resta quindi che identificare un sistema di gestione di basi di dati appartenente alla categoria degli ORDBMS al quale poter interfacciare l'applicazione web. A tale riguardo la scelta è caduta su PostgreSQL: un ORDBMS open source di classe enterprise.

2.2 - PostgreSQL, un ORDBMS open source di classe enterprise

La storia di PostgreSQL, che si pronuncia postgre-ES-CHIU-EL (SQL = "es chiu el"), parte nel lontano 1970 presso l'università della California, a Berkeley, dove iniziò lo sviluppo di un database relazionale chiamato Ingres. Alcuni anni dopo Ingres fu trasformato in un prodotto commerciale dalla società Relational Technologies, società che in seguito cambiò nome in Ingres Corporation, per poi essere successivamente acquisita dalla Computer Associates.

Nel 1986, il capo progetto di Ingres, Michael Stonebraker, che alcuni anni prima aveva lasciato il team di sviluppo di Berkeley per seguire la commercializzazione del prodotto, decise di ritornare all'accademia per dare vita ad un nuovo progetto post-Ingres. Il nuovo progetto aveva l'obbiettivo dichiarato di creare un prodotto che superasse gli evidenti limiti dei prodotti concorrenti dell'epoca. In particolare, puntava a fornire un supporto completo ai tipi di dati e la possibilità di definire nuovi tipi di dati (UDF, User Defined Types). Tutto ciò può essere riassunto dicendo che il team di sviluppo del nuovo progetto non fece altro che aggiungere delle caratteristiche object-oriented ad Ingres. Il nuovo progetto prese il nome di Postgres. Da segnalare che la base dei sorgenti di Ingres e di Postgres erano, e sono rimasti tutt'ora, ben distinti.

Come il suo predecessore anche Postgres fu ben presto commercializzato, questa volta per opera di una compagnia chiamata Illustra Information Technologies (poi parte della Informix Corporation), costituita da Paula Hawthorn, membro originale del team di Ingres, e dal sempre presente Michael Stonebraker. A differenza del suo antenato, Postgres non fu sviluppato commercialmente con la stessa rapidità di Ingres, infatti, dal 1988, anno in cui venne rilasciato il primo prototipo funzionante,

al 1993, furono sviluppate solo quattro versioni di Postgres, per poi giungere anche in questo caso alla terminazione del progetto.

Sebbene il progetto Postgres fosse ufficialmente abbandonato, la licenza BSD (Berkeley Software Distribution) dava comunque modo agli sviluppatori Open Source di ottenere una copia del software per poi migliorarlo a loro discrezione. Nel 1994 due studenti del Berkeley, Andrew Yu e Jolly Chen, aggiunsero a Postgres un interprete SQL, per rimpiazzare il vecchio linguaggio di interrogazione, QUEL, che risaliva ai tempi di Ingres. Il nuovo software venne quindi rilasciato sul web con il nome di Postgres95. Nel 1996 cambiò nome di nuovo: per evidenziare il supporto al linguaggio SQL, venne chiamato PostgreSQL. Il primo rilascio di PostgreSQL è stata la versione 6. Da allora ad occuparsi del progetto è una comunità di sviluppatori volontari provenienti da tutto il mondo che si coordina attraverso Internet. Alla versione 6 ne sono seguite altre, ognuna delle quali ha portato nuovi miglioramenti. Nel gennaio 2005 è stata rilasciata la versione 8. In questo lavoro di tesi verrà utilizzata la versione 8.1.5 di PostgreSQL. Nel momento in cui sto scrivendo l'ultima versione rilasciata dal team di sviluppo è la 8.2.3.

PostgreSQL ha sicuramente beneficiato della sua lunga storia: è assolutamente realistico affermare che oggi PostgreSQL è uno dei più avanzati database server presenti in circolazione. Riassumiamo di seguito alcune delle principali caratteristiche che si possono trovare in una distribuzione standard di PostgreSQL:

- **Object-Relational:** in PostgreSQL ogni tabella definisce una classe. PostgreSQL implementa l'ereditarietà fra tabelle, o fra classi se si preferisce. Funzioni e operatori sono polimorfici.

- **Conformità agli standards:** la sintassi utilizzata da PostgreSQL adotta la maggior parte dello standard SQL92 e molte caratteristiche presenti in SQL99. Alcune differenze di sintassi che possono essere trovate in PostgreSQL rappresentano delle caratteristiche uniche dello stesso PostgreSQL.

- **Open Source:** un team internazionale di sviluppatori si occupa di mantenere PostgreSQL. I membri del team vengono e vanno, ma i membri stabili hanno contribuito ad accrescere le potenzialità e le performance di PostgreSQL sino dal 1996. Un vantaggio della natura open source di PostgreSQL è che il talento e le conoscenze possono essere reclutati quando necessario. Il fatto che il team di sviluppo sia internazionale assicura che PostgreSQL è un prodotto che può essere utilizzato produttivamente in ogni linguaggio, non solo in inglese.

- **Transaction Model:** PostgreSQL protegge i dati e coordina gli accessi concorrenti multiutente attraverso le transazioni. Il "Transaction Model" utilizzato da PostgreSQL si basa sul modello MVCC (Multi-Version Concurrency Control). Tale modello provvede ad una migliore performance rispetto a quella riscontrabile in altri prodotti che coordinano la multiutenza attraverso i sistemi di bloccaggio a livello di tabella, di pagina o di riga.

- **Integrità referenziale:** PostgreSQL implementa completamente l'integrità referenziale supportando le relazioni tra chiavi esterne e primarie così come i triggers. Le "Business Rules" posso essere espresse all'interno del database piuttosto che contare su degli appositi strumenti esterni.

- **Linguaggi procedurali multipli:** triggers e altre procedure possono essere scritte in uno dei diversi linguaggi procedurali. Il codice lato server solitamente è scritto in PL/pgSQL, un linguaggio procedurale simile a Oracle PL/SQL. Il codice server-side può essere comunque sviluppato anche in Tcl, Perl e perfino in Bash, la shell open source Linux/Unix.

- **Supporto multiplo per APIs lato client:** PostgreSQL supporta lo sviluppo di applicazioni lato client in molti linguaggi. Ad esempio PostgreSQL può essere interfacciato da C, C++, Perl, PHP, Tcl/Tk e Python.

- **Tipi di dati unici:** PostgreSQL fornisce una vasta varietà di tipi di dati. Oltre agli usuali tipi numerici, stringa e data è possibile trovare tipi geometrici, tipi booleani e tipi di dati progettati specificatamente per occuparsi di indirizzi di rete.

- **Estensibilità:** una delle più importanti caratteristiche di PostgreSQL è che può essere esteso. Se non può essere reperito un qualcosa del quale si ha bisogno è possibile aggiungerlo. Per esempio è possibile aggiungere nuovi tipi di dati, nuove funzioni, nuovi operatori ed anche nuove procedure e linguaggi lato client. In riguardo ci sono molti pacchetti disponibili su Internet. Per esempio, Refractions Research Inc. ha sviluppato un insieme di tipi di dati geografici (GIS).

Da questo primo elenco di caratteristiche, dovrebbe essere immediato capire che non stiamo parlando di un prodotto simile agli altri database. Per quanto è stato possibile appurare fino ad ora è evidente che le enormi potenzialità di PostgreSQL derivano dal fatto che questo DBMS riesce a sfruttare in modo estremamente efficace ed efficiente la sua natura object-relational. Non è infatti operazione sempre semplice rendere compatibile il mondo SQL con quello della programmazione ad oggetti.
I database basati solo ed esclusivamente sul modello relazionale richiedono che sia l'utente a prelevare e raggruppare le informazioni correlate utilizzando le query SQL. Questo contrasta con il modo in cui, sia le applicazioni, che gli utenti, utilizzano i dati, come ad esempio avviene in un linguaggio di alto livello con tipi di dati complessi, dove tutti i dati correlati operano come elementi completi, normalmente definiti da oggetti o record (in base al linguaggio). Per creare un sistema che operi nelle modalità più vicine possibili a quello che può essere definito "il naturale utilizzo dei dati" non è però sufficiente creare un'infrastruttura basata sul semplice mix di SQL e di programmazione ad oggetti. Convertire le informazioni, dal mondo SQL a quello della programmazione orientata agli oggetti,

presenta infatti delle difficoltà dovute principalmente al fatto che i due mondi utilizzano modelli di organizzazione dei dati molto differenti. L'industria chiama questo problema “impedance mismatch” (discrepanza di impedenza): mappare i dati da un modello all'altro può assorbire fino al 40% del tempo di sviluppo di un progetto. Un certo numero di soluzioni di mappatura, normalmente dette “object-relational mapping”, possono risolvere il problema, ma tendono ad essere costose e ad avere delle controindicazioni: causando, ad esempio, scarse prestazioni oppure forzando tutti gli accessi ai dati ad aver luogo attraverso il solo linguaggio che supporta la mappatura stessa.

PostgreSQL può invece risolvere molti di questi problemi direttamente nel database. Esso permette agli utenti di definire nuovi tipi basati sui normali tipi di dato SQL, permettendo così al database stesso di comprendere dati complessi. Per esempio, si può definire un indirizzo come un insieme di diverse stringhe di testo per rappresentare il numero civico, la città, ecc. Da qui in poi si possono creare facilmente tabelle che contengono tutti i campi necessari a memorizzare un indirizzo con una sola linea di codice.

PostgreSQL, inoltre, permette l'ereditarietà dei tipi, uno dei principali concetti della programmazione orientata agli oggetti. Ad esempio, si può definire un tipo “codice_postale,” quindi creare un tipo “cap” (codice di avviamento postale) o un tipo “us_zip_code” (cap utilizzato in USA), basati su di esso. Gli indirizzi nel database potrebbero quindi accettare entrambi i tipi, e regole specifiche potrebbero validare i dati in entrambi i casi. Nelle prime versioni di PostgreSQL, implementare nuovi tipi richiedeva la scrittura di estensioni in C e la loro compilazione nel server dove era allocato il database. Dalla versione 7.4 è diventato molto più semplice creare ed usare tipi personalizzati attraverso il comando "CREATE DOMAIN".

Abbiamo anche menzionato, fra le caratteristiche di PostgreSQL, la possibilità di rendere disponibili nel server nuove funzioni e nuovi operatori. La maggior parte dei sistemi RDBMS permette agli utenti di scrivere una procedura, ovvero un blocco di codice SQL che le altre istruzioni SQL possono richiamare. Comunque il SQL stesso rimane inadatto come linguaggio di programmazione, pertanto gli utenti

possono sperimentare grandi difficoltà nel costruire logiche complesse. Ancora peggio, il SQL stesso non supporta molti dei principali operatori di base dei linguaggi di programmazione, come le strutture di controllo di ciclo e condizionale. Pertanto, ogni venditore per aggiungere queste caratteristiche ha scritto le sue estensioni al linguaggio SQL, ed inoltre, queste estensioni, non sono quasi mai costruite per operare su piattaforme diverse di database. In PostgresSQL questo problema non sussiste, poiché i programmatori possono implementare la logica in uno dei molti linguaggi supportati. Il programmatore può inserire il codice sul server come funzioni, che rendono il codice riutilizzabile come stored procedure; così facendo, il codice SQL può richiamare funzioni scritte in altri linguaggi (come il C o il Perl). Non meno importante è la presenza di un linguaggio nativo chiamato PL/pgSQL, simile al linguaggio procedurale di Oracle PL/SQL, che offre particolari vantaggi nelle procedure che fanno un intensivo uso di query. Questo linguaggio sarà tra l'altro il linguaggio in cui verrà implementata la funzione di confronto tra gli istogrammi delle immagini.

Queste caratteristiche fanno di PostgreSQL, probabilmente, il più avanzato sistema di gestione dei dati dal punto di vista della programmabilità. Utilizzare PostgreSQL può ridurre fortemente il tempo totale di programmazione di molti progetti, con i vantaggi suddetti che crescono con la complessità del progetto stesso.

A questo punto non ci resta che iniziare ad utilizzare PostgreSQL e le sue potenzialità per creare un database che possa supportare le caratteristiche dell'applicazione web che deve essere realizzata. Nel prossimo paragrafo quindi ci dedicheremo ad analizzare gli aspetti di PostgreSQL utili alla definizione del database.

2.3 - La strutturazione del database

Da questo momento iniziamo ad affrontare in maniera dettagliata le scelte da intraprendere per strutturare il database, che dovrà essere progettato in modo tale da poter supportare le funzionalità richieste dall'applicazione web che deve essere

realizzata.

2.3.1 - La memorizzazione delle immagini

La prima situazione da analizzare riguarda la memorizzazione delle immagini. Risulta evidente, per quanto fino ad ora appurato, che l'applicazione web in questione deve essere in grado di fare sostanzialmente due cose: archiviare in modo permanente delle immagini e permettere di effettuare delle ricerche tra le immagini memorizzate mediante confronti. Abbiamo anche appreso, nel precedente capitolo, che di fatto, l'oggetto di tali confronti non sarà il codice sorgente dell'immagine, ma bensì il suo istogramma di colore, una struttura che permette la rappresentazione del contenuto cromatico di un'immagine. Se può essere lecito pensare, alla luce di quanto è stato fatto intendere fino a questo momento, che le immagini dovranno essere memorizzate nel database, è altrettanto legittimo effettuare un'attenta analisi sui risvolti che questa scelta potrebbe implicare e se questa sia effettivamente la scelta che rende l'applicazione più performante. Ciò non significa sconfessare quanto è stato fin qui sostenuto, ma effettuare un'analisi sulle possibili modalità con le quali procedere alla memorizzazione delle immagini è un atto doveroso, soprattutto se vengono prese in dovuta considerazione delle situazioni oggettive direttamente connesse alla natura ed allo sviluppo dell'applicazione web, cosa che intendiamo fare nell'immediato proseguo di questo paragrafo. Poiché questa considerazione potrebbe minare quella che fino ad ora poteva essere considerata una certezza, ovvero il fatto di archiviare le immagini tramite l'utilizzo del database per la memorizzazione del loro codice sorgente, cerchiamo di spiegare meglio in base a quali parametri viene rimesso in discussione questo fatto.

Abbiamo detto nell'introduzione che i sistemi per la gestione dei dati offrono già, mediante l'utilizzo di campi di tipo BLOB e CLOB, la possibilità di memorizzare dati di tipo strutturato, come le immagini. Abbiamo inoltre messo in evidenza, proprio alla luce di ciò, che lo scopo primario di questa tesi di ricerca non deve essere individuato nella memorizzazione delle immagini in un database, bensì

nell'offrire la possibilità di confrontare queste immagini tramite un opportuno predicato da inserire in una dichiarazione SQL, dove, di fatto, si fa esplicitamente riferimento ad una struttura che qualifica un aspetto del contenuto delle immagini. In questo caso, come abbiamo avuto già modo di dire, faremo uso a tal fine dell'istogramma (o descrittore) di un'immagine. In conclusione, quindi, analizzare quelle che possono essere le varie strade percorribili per memorizzare in modo permanente le immagini, non deve essere considerato come una "marcia indietro", anche se in tale analisi verranno contemplate soluzioni che non prevedono l'utilizzo attivo del database. Questo processo deve essere visto invece come un'azione ovvia, volta ad individuare la soluzione più performante per la creazione ed il funzionamento dell'applicazione. Da ora in avanti, nel proseguo della trattazione, daremo sempre per appurato che il "cuore della ricerca" di questo lavoro di tesi è localizzato nella memorizzazione nel database di una struttura ausiliaria che rappresenti il contenuto cromatico di un immagine, struttura che di fatto è a tutti gli effetti un dato di tipo strutturato e che verrà sempre utilizzata in ogni richiesta di confronto tra immagini. Compreso questo scenario, non deve essere quindi vista come una negazione di quanto argomentato fino ad ora, il fatto di analizzare la possibilità di memorizzare le immagini senza l'utilizzo attivo del database, se questa scelta dovesse poi rendere l'applicazione più performante.

Prima di analizzare in dettaglio pro e contro delle singole soluzioni di memorizzazione delle immagini, è necessario puntualizzare due aspetti riguardo alle caratteristiche dell'applicazione, aspetti da tenere ben presenti, poiché abbiamo appena evidenziato che questi saranno influenti ai fini della scelta del sistema di memorizzazione da adottare. Il primo aspetto riguarda la mappatura delle immagini: è stato appurato, per motivi già noti, che ogni singola immagine inserita nel sistema deve essere mappata; questo comporta che per ogni immagine inserita nel sistema saranno fisicamente presenti due file, o comunque, due sorgenti binari distinti, ai quali ricondurre l'immagine originale e quella mappata. Il secondo aspetto riguarda invece le caratteristiche dell'interfaccia dell'applicazione: è previsto che l'applicazione consenta di raggruppare le immagini inserite in gallerie e che ad ogni

galleria si possa accedere per prendere visione delle singole immagini che essa contiene. Per motivi di layout è prassi comune delle applicazioni web presentare l'insieme delle immagini presenti in ogni galleria sotto forma di preview, realizzato mediante la creazione di un thumbnail per ogni immagine di dimensioni fissate. Inoltre è altresì previsto che l'applicazione fornisca all'utente la possibilità di visionare sia il thumbnail dell'immagine originale che quello dell'immagine mappata. Tutto ciò significa che per ogni immagine inserita nel sistema dovranno essere presenti non due, ma ben quattro sorgenti binari distinti: quello dell'immagine originale, quello dell'immagine mappata, quello del thumbnail dell'immagine originale e quello dell'immagine mappata. Infine è importante sottolineare che, affinché l'applicazione web possa visualizzare le immagini, originali o mappate, full size o thumbnail che siano, è sempre necessaria la presenza sul disco dei file che rappresentano l'immagini, ai quali deve sempre essere possibile far riferimento specificando il loro path fisico nell'attributo “src” del tag <img />.

Fatte queste doverose considerazioni, possiamo passare all'analisi delle possibilità di memorizzazione, tenendo ben presente quanto appena detto, poiché, ripeto di nuovo, ciò influisce direttamente sulla scelta della modalità di memorizzazione che verrà fatta. Dall'analisi effettuata, esistono tre distinte possibilità per memorizzare in modo permanente un'immagine e renderla disponibile ad un'applicazione di tipo web:

1. Memorizzazione dell'immagine nel database in un campo di tipo BLOB.
2. Memorizzazione dell'immagine nel database utilizzando un campo di tipo OID.
3. Memorizzazione dell'immagine nel disco mediante l'utilizzo del filesystem, specificando un riferimento ad essa nel database tramite un campo testo.

La prima soluzione prevede la definizione di un campo di tipo BLOB da utilizzare per memorizzare l'immagine. PostgreSQL offre a tale scopo il tipo BYTEA. Per

PostgreSQL un campo di tipo BYTEA è deputato alla memorizzazione di "*raw data*". Con questa terminologia s'intendono classificare dei dati la cui struttura ed il cui significato non è compreso dal database. PostgreSQL, infatti, per gli altri tipi di dati, quali ad esempio INTEGER o CHARACTER, sa che i byte posti in una colonna dichiarata di tipo INTEGER sono supposti rappresentare un valore intero, sul quale effettuare tutte le operazioni del caso quando si manipola un intero. Diversamente, per un campo BYTEA, PostgreSQL non inferisce alcun significato ai dati: li considera solo come una collezione di bit senza preoccuparsi di cosa essi realmente rappresentino! Analizzando quindi questa possibilità sorge un primo problema dovuto alle modalità con cui PostgreSQL richiede l'inserimento dei dati in un campo di tipo BYTEA: la sintassi prevede un inserimento identico a quello utilizzato per la memorizzazione delle stringhe, ovvero la delimitazione dell'insieme di caratteri fra singoli apici. Il fatto poi che questi caratteri siano dei bit che rappresentano il contenuto di un documento di testo o di un immagine non è di interesse per il DBMS, ma se ne dovrà preoccupare l'applicazione che poi li utilizzerà! Questo significa che per memorizzare un immagine in un campo di questo tipo è assolutamente necessaria un'applicazione esterna che si preoccupi di estrarre dall'immagine il suo codice sorgente. Pur sapendo che questa operazione è comunque necessaria per ricavare l'istogramma dell'immagine, nel contesto dell'applicazione web, non è sicuramente la situazione più performante quella in cui non è immediatamente disponibile sotto forma di file il codice dell'immagine. Abbiamo messo in risalto, infatti, che per essere visualizzate da un browser le immagini devono essere fisicamente presenti sul disco. Inoltre è stato anche illustrato che per ogni immagine inserita, di fatto, l'applicazione deve gestire quattro immagini distinte, che devono essere sempre riconducibili ad altrettanti file distinti. Tutto ciò implica che l'eventuale scelta di memorizzare le immagini in un campo BYTEA comporta, in realtà, l'utilizzo di quattro campi distinti di questo tipo, uno per ogni immagine. A questo punto è immediato il prendere atto del fatto che l'applicazione sarebbe estremamente poco performante se, ad ogni richiesta di visualizzazione di un'immagine, dovesse essere costretta a richiedere l'intervento di

uno strumento esterno al database che, per prima cosa, deve interpretare i dati presenti nei campi BYTEA e, successivamente, andare a creare i file temporanei sul disco ai quali fare riferimento per la visualizzazione delle immagini nel browser. Come avremo modo di comprendere quando parleremo delle GD2, le librerie grafiche di PHP utilizzate per manipolare le immagini, ciò significherebbe che per visualizzare una galleria costituita, ad esempio, da 20 immagini, sarebbe necessaria la creazione di 80 immagini "resource" residenti in memoria principale, dalle quali creare in seguito 80 file temporanei sul disco. Inutile sottolineare che, implementare la visualizzazione tramite un'infrastruttura di questo tipo, genera un sovraccarico alla funzionalità dell'applicazione che non ha nessun vantaggio. In definitiva, quindi, questa soluzione non sembra essere quella più adatta, o meglio, il fatto di dover avere necessariamente, ai fini della visualizzazione, il codice dell'immagine in un file fisicamente residente sul disco fisso rende questa soluzione poco performante allo scopo.

La seconda soluzione è quella che prevede l'utilizzo di un campo di tipo OID (Object Identifier). PostgreSQL è un ORDBMS, ed abbiamo già discusso del fatto che per ciò è assolutamente lecito vedere una relazione (tabella) come una classe, ed ogni riga (tupla) della tabella come l'istanza di tale tabella e quindi anche della classe; di conseguenza una riga può essere considerata di fatto un oggetto. In ragione di tutto questo ogni tabella prevedeva, fino alla versione 8.0 di PostgreSQL, un campo nascosto di tipo OID, nel quale era mantenuto un identificativo univoco di tale riga (oggetto); tale identificativo era univoco per tutto il database ed in futuro il team di sviluppo aveva previsto di renderlo unico per ogni singola tabella. Attualmente, invece, c'è stata un'inversione di tendenza, poiché l'attributo OID non viene più inserito di default nelle tabelle, ma solo se esplicitamente richiesto. Detto questo vediamo come tutto ciò potrebbe essere utilizzato per memorizzare un'immagine.

PostgreSQL supporta l'utilizzo dei così detti "*large_objects*". Un "*large_object*" è una collezione di bit che può superare la dimensione massima stabilita per un campo BYTEA, pari ad 1GB. Oltre a questo fatto, la principale differenza da

quest'ultimo tipo risiede nel fatto che un "*large_object*" non viene memorizzato nel campo OID della tabella di competenza. Un "*large_object*" viene fisicamente memorizzato in una tabella di sistema chiamata "*pg_largeobject*". Nella tabella di competenza viene quindi memorizzato l'OID assegnato alla riga della tabella "*pg_largeobject*" che effettivamente contiene il codice del "*large_object*". PostgreSQL offre le funzioni "*lo_import('file_path')*", "*lo_export(OID,'file_path')*" e "*lo_unlink(OID)*" che rispettivamente importano, esportano e cancellano un "*large_object*" dalla tabella interessata e quindi dal database stesso. Ovviamente, come spero possa essere già stato intuito, un "*large_object*" può essere un'immagine. Questa soluzione, se analizzata con attenzione, seppur in forma diversa, presenta i soliti problemi di quella precedente. L'unica differenza è dovuta al fatto che per rendere disponibili i file delle immagini sul disco, invece che richiedere l'intervento delle librerie GD2, verrebbero utilizzate le funzioni citate poche righe sopra atte alla manipolazione dei "*large_object*". Inoltre, le soluzioni che prevedono un riferimento tramite OID, soffrono comunque di due problemi contingenti: uno relativo al backup e l'altro quando si tenta di utilizzare l'OID per identificare una riga al fine di una ricerca (SELECT). Nel primo caso il problema è dovuto al fatto che quando si effettua un restore dal backup di un database, il sistema è solito non riassegnare alle righe gli stessi OID (specificando sempre in fase di dump che debbano essere generati) e risulta quindi necessario, per mantenere la consistenza dei dati, prendere degli accorgimenti che non sono contemplati di default dalla funzione "*pg_dump*" di PostgreSQL. Nel secondo, invece, si deve considerare che un campo OID occupa 32 bit, ragion per cui solo nel caso in cui la PRIMARY KEY della relazione è veramente molto lunga può aver senso, in termini di prestazioni, effettuare delle ricerche utilizzando per riferimento l'OID delle righe. Tutto ciò, unito alla constatazione che in un certo senso gli OID sono stati deprecati dal team di sviluppo di PostgreSQL, invita a prendere atto del fatto che anche questa soluzione non è adatta allo sviluppo di un'applicazione di natura web.

L'ultima soluzione è quella che prevede l'intervento del filesystem per memorizzare un'immagine. In pratica nella tabella preposta alla memorizzazione delle immagini

non viene memorizzato il suo codice sorgente, ma i riferimenti necessari ad individuarla sul disco, come il suo nome ed il suo path. Chi si occupa di memorizzare fisicamente l'immagine sul disco del server (non nel database), è il filesystem. Questo può essere effettuato mediante una chiamata ad una routine presente sul server, scritta ad esempio in PHP, oppure in C o in qualsiasi altro linguaggio che consenta questa operazione. Lo scenario che si prospetta è quindi il seguente: il sistema riceve una richiesta di upload di un'immagine e tramite il filesystem provvede alla sua memorizzazione nel disco fisso. In realtà, come già sappiamo, per ogni upload, le immagini memorizzate saranno quattro e non una sola. Conseguenza diretta di questo fatto è che la tabella preposta alla memorizzazione delle immagini dovrà essere strutturata in modo tale da mantenere i riferimenti ad ognuna di queste quattro immagini. Adottando questa soluzione vengono quindi eliminati i problemi di sovraccarico che sorgerebbero, per i motivi già esposti, in fase di visualizzazione di una galleria, qualora le immagini fossero memorizzate utilizzando una delle altre due soluzioni.

2.3.2 - La memorizzazione dell'istogramma di colore.

A questo punto, anche se sono state date in precedenza delle ampie spiegazioni in riguardo, potrebbe comunque sorgere una domanda spontanea: quale è il valore aggiunto, del quale risulta privo un normale Relational Data Base Management System, presente invece in PostgreSQL che consente di realizzare l'infrastruttura fino ad ora descritta? La risposta è nessuno, ma se questa trattazione è stata letta con attenzione, mi auguro che ciò non sia motivo di stupore! Sarebbe stato motivo di stupore se avessimo ipotizzato di sviluppare il criterio di confronto tra le immagini andando a confrontare i codici sorgenti delle stesse; in tal caso, a questo punto, sarebbe stato più che lecito pensare che qualcosa non funziona come dovrebbe, poichè non avrebbe avuto alcun senso utilizzare come oggetto del confronto i codici sorgenti delle immagini e strutturare il database in un modo tale da non avere i

medesimi disponibili nel database stesso. Sappiamo però che la situazione nella quale ci troviamo non è questa, poiché per effettuare i confronti verrà utilizzato il descrittore delle immagini e non il loro codice sorgente. Per realizzare il confronto sarà quindi necessaria la presenza nel database dei descrittori, di questo dato strutturato che descrive il contenuto cromatico di un immagine. A tale riguardo entra adesso in gioco tutta la potenza di PostgreSQL, il quale consente di dichiarare un campo di una tabella di tipo vettore e di definire un'opportuna funzione che manipoli i vettori, che può essere richiamata nella dichiarazione di una query SQL, possibilità che non è offerta dai normali RDBMS.

Diciamo subito che dalla versione 7.4 (in modo parziale), alla versione 8.1.5 (in modo completo), utilizzata per questo lavoro di tesi, PostgreSQL fornisce il supporto al tipo array in ogni sua distribuzione. Tanto per fare un esempio il seguente codice SQL è perfettamente eseguito dall'ORDBMS:

```
CREATE TABLE prova_array(
    id SERIAL,
    vettore INTEGER [10]);

INSERT INTO prova_array VALUES(
    nextval('prova_array_id_seq'),
    ARRAY[1,2,3,4,5,6,7,8,9,10]);
```

PostgreSQL ha una gestione molto flessibile del tipo array. Oltre al precedente inserimento, mediante la forma chiamata "*constructor syntax*", è lecita anche la seguente sintassi:

```
INSERT INTO prova_array VALUES(
    nextval('prova_array_id_seq'),
    '{1,2,3,4,5,6,7,8,9,10}';
```

Sempre in termini di flessibilità è giusto menzionare il fatto che non c'è un limite da rispettare per il numero di membri che un array può contenere. Il numero di membri specificati nella dichiarazione di un campo array, dieci nell'esempio, non è

vincolante, infatti non è fonte di alcun errore effettuare un inserimento di questo tipo:

```
INSERT INTO prova_array VALUES(
    nextval('prova_array_id_seq'),
    ARRAY[1,2,3,4,5,6,7,8,9,10,11]);
```

Ciò non significa che sia "buona cosa" utilizzare gli array senza aver ben presente quanti elementi effettivamente contengono: a tal proposito, comunque, quando necessario è possibile ricavare tale informazione tramite la funzione "*array_dims()*", inserita in una query SQL nel modo seguente:

```
SELECT array_dims(vettore)
FROM prova_array
WHERE id=1;
```

Tale necessità non dovrebbe comunque verificarsi nel contesto dell'applicazione che verrà realizzata, poiché è sempre noto a priori quanti sono gli elementi che costituiscono l'istogramma dell'immagine.

Abbiamo visto quindi come PostgreSQL viene incontro alle esigenze dell'applicazione che dobbiamo costruire mettendo a disposizione il tipo array. In un contesto nel quale non fosse stato possibile la memorizzazione in un campo di una tabella di un dato strutturato di questo genere, la creazione dell'applicazione nelle modalità desiderate sarebbe divenuta assai complessa, per non dire impossibile. In particolare sarebbe stato impossibile effettuare un qualsiasi confronto tramite l'inserimento di un opportuno predicato nella clausola "WHERE" di una dichiarazione SQL, poiché, per memorizzare l'istogramma di un'immagine, si sarebbe dovuto procedere alla definizione di un'apposita relazione, caratterizzata da una chiave primaria, da una chiave esterna riferita alla chiave primaria della tabella delle immagini e da un campo "percentuale di colore" di tipo reale, dove memorizzare ogni singolo valore presente nel descrittore di una qualsiasi immagine

memorizzata. Così facendo, è ovvio che il confronto tra due istogrammi, riferiti a due immagini distinte, sarebbe stato possibile solo con l'ausilio di un linguaggio di alto livello interfacciato al DBMS utilizzato. Lo scopo di questa tesi di ricerca è proprio quello di superare questa limitazione. L'obbiettivo non è quello di eseguire una "SELECT" per recuperare i valori di due istogrammi dal database ed eseguire il confronto tra essi utilizzando i costrutti e le funzionalità messe a disposizione, ad esempio, da PHP. L'obbiettivo è quello di effettuare il confronto richiamando un'operatore di confronto tra campi di tipo array, appositamente definito per tale scopo tramite il linguaggio PL/pgSQL, direttamente nella clausola "WHERE" della dichiarazione SQL. Il risultato della "SELECT" dovrà essere quindi l'ID dell'immagine (o gli ID delle immagini in caso di confronto multiplo) se l'operatore di confronto dà esito positivo, l'insieme vuoto altrimenti.
Prima di affrontare l'implementazione dell'operatore di confronto è senza dubbio auspicabile aver presenti le definizioni delle tabelle che costituiscono il database. Il prossimo paragrafo sarà quindi dedicato a soddisfare tale auspicio.

2.3.3 - La definizione delle tabelle del database

Siamo quindi giunti alla definizione delle tabelle che andranno a costituire il database. Le tabelle saranno tre: una per la memorizzazione delle immagini, nella modalità esposta, una per le gallerie ed una per i coefficienti della matrice di similarità, della quale abbiamo parlato nel precedente capitolo. Delle immagini abbiamo già affrontato alcuni aspetti sostanziali, dalla cui analisi dovremmo aver capito, almeno in parte, la natura di alcuni dei campi che costituiranno la tabella. In particolare sappiamo per certo la tipologia di cinque campi: i quattro nei quali vengono memorizzati i nomi delle immagini, dichiarati di tipo VARCHAR, e quello preposto alla memorizzazione del descrittore, che come abbiamo appreso è un tipo ARRAY, i cui elementi sono dichiarati di tipo NUMERIC, poiché sappiamo che il contenuto cromatico è espresso in percentuale. Possiamo a questo punto mostrare la definizione dell'intera tabella e procedere in seguito alla descrizione dei singoli

campi che la costituiscono:

```
CREATE TABLE immagini (
        id_immagine SERIAL PRIMARY KEY,
        nome VARCHAR(50) NOT NULL,
        nome_thumb VARCHAR(50) NOT NULL,
        mapped VARCHAR(50) NOT NULL,
        mapped_thumb  VARCHAR(50) NOT NULL,
        formato CHAR(4) NOT NULL,
        descrittore NUMERIC[122] NOT NULL,
        larghezza SMALLINT NOT NULL,
        altezza SMALLINT NOT NULL,
        dimensione INTEGER NOT NULL,
        aspect_ratio VARCHAR(9) NOT NULL,
        didascalia VARCHAR(100),
        caratteristiche TEXT,
        galleria INTEGER NOT NULL REFERENCES gallerie(id_galleria)
                                  ON UPDATE CASCADE
                                  ON DELETE CASCADE
);
```

Il primo campo “id_immagine” rappresenta la chiave primaria della relazione “immagini”, come del resto è ben evidente nella definizione della tabella. Riguardo a questo campo deve essere prestata attenzione al suo tipo: SERIAL. Questo tipo è una particolarità offerta da PostgreSQL: una colonna dichiarata di tipo SERIAL è di fatto un INTEGER senza segno, il cui valore viene automaticamente incrementato (o decrementato) di un valore prestabilito (pari ad un'unità di default) ad ogni nuovo inserimento di riga. Volendo fare un breve excursus, possiamo affermare che questo campo si comporta apparentemente come un campo dichiarato INTEGER AUTO_INCREMENT presente in una tabella creata con MySQL, un diffusissimo DBMS open source utilizzato prevalentemente in applicazioni di tipo web. La differenza tra quest'ultimo ed un tipo SERIAL di PostgreSQL risiede nel fatto che MySQL, per un campo AUTO_INCREMENT, si limita a recuperare l'ultimo valore inserito nel campo e lo incrementa di una unità, mentre PostgreSQL, quando incontra un campo di tipo SERIAL provvede alla creazione di una SEQUENCE. Una SEQUENCE è un oggetto che genera in maniera automatica una sequenza di numeri unici. Questo è il metodo più sicuro per avvalorare un campo dichiarato

PRIMARY KEY per le entità alle quali non può essere associato per natura un ID unico, come, ad esempio, può essere ritenuto il codice fiscale in un'ipotetica relazione “persona”. Esistono delle funzioni predefinite che operano con le SEQUENCE, tra le quali possiamo segnalare la funzione “nextval('immagini_id_immagine_seq')”, utilizzata in automatico ad ogni nuovo inserimento di riga nella tabella “immagini”. L'argomento della funzione è la SEQUENCE creata da PostgreSQL subito dopo la creazione della tabella, creazione che avviene automaticamente ogni qual volta PostgreSQL riceve la dichiarazione di un campo SERIAL in una qualsiasi tabella. La funzione provvede a generare il valore successivo per la sequenza specificata. L'uso di un tipo SERIAL per un campo dichiarato PRIMARY KEY assicura quindi che il valore dell'ID ad esso assegnato sia sempre unico, senza così rischiare inserimenti scorretti che non andrebbero a buon fine per mancato rispetto del vincolo imposto.

Dei quattro campi successivi abbiamo già esposto cosa essi rappresentano, ovvero i nomi dei file, comprensivi di estensione, che vengono memorizzati sul disco in seguito ad ogni singola operazione di upload di un'immagine. Il campo “formato” memorizza l'estensione del file, anche se questa, abbiamo appena detto essere comunque inserita nei nomi stessi dei file. I possibili valori di questo campo sono quattro: gif, jpg, jpeg e png, ovviamente corrispondenti ai quattro formati d'immagine supportati dai browser web. Per quanto riguarda il campo “descrittore”, anche per esso abbiamo già appurato la sua fondamentale importanza, poiché è il campo intorno al quale ruota la principale funzionalità dell'applicazione, cioè il confronto tra immagini. I campi “larghezza” ed “altezza” sono preposti a quanto è facilmente intuibile: memorizzare la dimensione orizzontale e verticale dell'immagine, espressa in numero di pixel. Il campo “dimensione” indica invece il numero di Kbyte occupati dal file dell'immagine originale sul disco. “Aspect_ratio” è un campo di tipo stringa che può assumere tre distinti valori: 'LANDSCAPE', 'PORTRAIT' e 'SQUARE', i quali indicano rispettivamente che l'immagine è sbilanciata in larghezza, sbilanciata in altezza oppure quadrata. I campi “didascalia” e “caratteristiche” sono dei campi di appoggio nei quali memorizzare un breve

commento sull'immagine ed un approfondimento sulle eventuali caratteristiche della medesima; da notare che l'assenza del vincolo NOT NULL indica che il loro inserimento è facoltativo. Troviamo infine il campo “galleria”, il quale ha un vincolo di integrità referenziale con la chiave primaria della relazione “gallerie”. La dichiarazione di questo vincolo è ovvia conseguenza del fatto che ogni immagine inserita deve essere collocata in una galleria. Inoltre, è da notare che l'eventuale aggiornamento o cancellazione di una galleria determina una corrispondente azione in questo campo. In particolare, deve essere evidenziato che in caso di cancellazione di una galleria, la politica adottata dal sistema impone che tutte le immagini contenute in essa vengano conseguentemente cancellate. Un'ultima annotazione da fare riguarda il fatto che in precedenza, parlando delle modalità di memorizzazione delle immagini, avevamo affermato, relativamente alla scelta fatta, che nella tabella “immagini” andavano memorizzate le informazioni necessarie per individuare i file sul disco corrispondenti ad ogni singola istanza di tale tabella, mentre è possibile notare, che tale tabella annovera solo i nomi delle immagini, ma nessun riferimento al path di sistema. In realtà, tramite il campo “galleria”, è sempre possibile risalire al path delle immagini, poiché ogni galleria è fisicamente caratterizzata da una directory sul disco che contiene tutte le immagini ad essa associate e tutte le directory sono collocate in una posizione del disco univocamente individuabile, prendendo come riferimento la root directory che contiene l'applicazione, la quale è ovviamente nota. Passiamo ora alla definizione della tabella “gallerie”:

```
CREATE TABLE gallerie (
        id_galleria SERIAL PRIMARY KEY,
        nome VARCHAR(50) NOT NULL,
        max_img SMALLINT NOT NULL,
        descrizione VARCHAR(100),
        caratteristiche TEXT
);
```

Come nel caso della tabella “immagini”, anche in questa, il campo “id_galleria” dichiarato PRIMARY KEY, è di tipo SERIAL; non ripetiamo quindi i motivi di questa scelta, poiché sono esattamente gli stessi esposti per l'altra tabella. Il campo

“nome”, di tipo VARCHAR, è una stringa che memorizza semplicemente il nome assegnato alla galleria. Il campo “max_img” indica il numero massimo di immagini che la galleria può contenere, impostato di default a 30 immagini. Infine, i campi “descrizione” e “caratteristiche” espletano la funzione ausiliaria di memorizzare una breve descrizione della galleria e le sue eventuali caratteristiche; da notare che il loro inserimento non è necessariamente richiesto. Un esempio di utilizzo di tali campi potrebbe essere quello relativo ad una galleria con nome='MONTAGNE BIANCHE', descrizione='Galleria di fotografie scattate sulle Dolomiti' e caratteristiche='Galleria che contiene immagini con elevata presenza di bianco, azzurro e toni di grigio'. Il fatto di raggruppare le immagini in gallerie tematiche nasce dal presupposto che i confronti hanno un senso quando vengono effettuati tra immagini che descrivono scenari simili o almeno compatibili. Non è ipotizzabile che un eventuale confronto, tra un'immagine raffigurante delle montagne innevate con l'immagine di un auto d'epoca, possa avere una qualche valenza logica e fornisca un risultato significativo. Questo non significa che il confronto non possa dare esito positivo anche in una situazione del genere, sappiamo infatti che l'istogramma descrive il contenuto cromatico di un immagine ed è tale contenuto che determina l'uguaglianza o la similarità tra immagini; a tale riguardo è stato fatto anche un esempio pratico del paragrafo 1.4 del primo capitolo. Nonostante questo, si ritiene, però, che sia abbastanza naturale cercare di porsi in un contesto nel quale i confronti possano essere significativi, per questa ragione è stato deciso di inserire le immagini in delle gallerie che le raggruppano in base al contesto che le immagini stesse raffigurano.

Resta adesso da mostrare la definizione della terza tabella, quella nella quale vengono memorizzati i coefficienti della matrice di similarità:

```
CREATE TABLE pesi (
        id_peso SERIAL PRIMARY KEY,
        peso NUMERIC NOT NULL
);
```

La struttura di tale tabella è molto semplice: abbiamo infatti il campo “id_peso” dichiarato come PRIMARY KEY, sempre di tipo SERIAL, ed il campo “peso”, di tipo NUMERIC, preposto alla memorizzazione dei coefficienti. Sapendo che questi coefficienti sono ben rappresentati tramite una struttura di tipo matriciale, forse si attendeva, ai fini della loro memorizzazione in una tabella, la definizione di un campo di tipo ARRAY multidimensionale. In realtà, quella appena descritta è stata la prima scelta fatta, che però, come avremo modo di capire tra breve, non si è rivelata adatta per l'utilizzo dei coefficienti all'interno di una funzione scritta in PL/pgSQL

Avendo adesso ben presente la struttura delle tabelle che costituiscono il database, possiamo passare al “cuore dell'applicazione”, ovvero, all'implementazione della funzione di confronto tra immagini.

2.4 - La funzione di confronto tra immagini

Siamo finalmente arrivati al momento di implementare la funzione della quale, più o meno, si è parlato in ogni singolo paragrafo di questa tesi. La realizzazione di una funzione che consenta di confrontare il contenuto cromatico di due o più immagini, attraverso l'utilizzo di un database è, infatti, il cardine di questa tesi di ricerca.

Di tale funzione sono stati già esposti diversi aspetti. Sappiamo di sicuro che l'input è rappresentato dagli istogrammi delle due immagini che vogliono essere confrontate; oramai è anche noto che il risultato della funzione è un valore con il quale poter quantificare quanto questi due istogrammi sono simili. Inoltre, per valutare questa similarità tra i contenuti cromatici delle immagini, tenendo conto della naturale percezione cromatica dei colori, abbiamo introdotto la matrice di similarità, della quale a suo tempo è stato spiegato in che modo fornisca un valido supporto a tal fine. Resta quindi adesso da identificare in che modo elaborare questi dati, per determinare il valore d'interesse, che da ora in poi chiameremo “distanza”, con il chiaro fine di indicare esplicitamente che questo valore è di fatto una misura della distanza che intercorre tra due istogrammi. Per realizzare la funzione, è stato

fatto esplicito riferimento un articolo sullo sviluppo di progetti QBIC (Query By Image Content), nel quale vengono proposti alcuni dei possibili metodi di ricerca di immagini per contenuto di colore, basati sul calcolo di una distanza pesata tra istogrammi. La formula utilizzata per questo calcolo è la seguente:

$$Z_k = H1_k - H2_k \forall k = 1..N$$

$$M_{i,j} = \sum_{i=1}^{N} \sum_{j=1}^{N} Z_i \times Z_j \times M_{i,j}$$

Prospetto 2.1 - Formula utilizzata per calcolare la distanza tra istogrammi

Il risultato prodotto da questa formula prende appunto il nome di "Histogram Distance", volendo appunto indicare che il valore prodotto rappresenta una forma, quadrata in questo specifico caso, di distanza tra i due istogrammi considerati. Inoltre, con la scelta fatta per determinare i coefficienti della matrice di similarità (vedi Capitolo 1, paragrafo 1.4), è assicurato che il risultato sia sempre un valore non negativo. Non riportiamo in questa sede la dimostrazione di questo asserto poiché ciò esula dagli scopi di questa tesi.

Abbiamo adesso tutti gli elementi per creare una funzione che, utilizzando la formula appena introdotta, fornisca come risultato la distanza tra due generici istogrammi. Come annunciato, la funzione verrà implementata nel linguaggio procedurale server side PL/pgSQL di PostgreSQL. Questo è uno dei linguaggi procedurali che PostgreSQL ha a disposizione per estendere le funzionalità del server tramite la realizzazione di stored procedures. Generalmente, durante la fase d'installazione di PostgreSQL, l'attivazione di questo linguaggio è opzionata di default; comunque sia, è sempre possibile verificare se il linguaggio è attivo tramite la query "SELECT * FROM pg_language;". In caso negativo, il linguaggio può essere abilitato semplicemente tramite l'esecuzione dalla shell DOS di Windows del

comando “CREATELANG plpgsql database-name”.

PL/pgSQL è un linguaggio che combina la natura dichiarativa dei comandi SQL con le strutture solitamente offerte da un qualsiasi linguaggio di programmazione. Una funzione scritta in PL/pgSQL: permette la dichiarazione di variabili e l'utilizzo di costrutti iterativi e condizionali; accetta parametri in ingresso; consente di richiamare al suo interno altre funzioni; permette la dichiarazione di una qualunque query SQL e la manipolazione dei dati restituiti dall'esecuzione della query. A margine di quest'ultima caratteristica è giusto sottolineare da subito che questo linguaggio va identificato come un linguaggio SQL-based, al quale sono state aggiunte le principali caratteristiche appartenenti ad un qualsiasi linguaggio di programmazione, e non viceversa. Questa considerazione deve essere tenuta ben presente, poiché da essa deriva direttamente il fatto che PL/pgSQL esprime il massimo delle sue performance quando è chiamato ad un intensivo uso di query SQL e non quando deve eseguire operazioni che possono essere svolte da un qualunque linguaggio di programmazione. Per creare una funzione efficiente in PL/pgSQL, è quindi necessario ragionare sempre in un'ottica orientata all'SQL.

In principio, io non ho avuto questa accortezza, e posso affermare che le conseguenze di un approccio sbagliato, all'implementazione delle funzioni scritte in PL/pgSQL, possono avere delle ripercussioni devastanti sulle prestazioni offerte dalle query che ne fanno uso. Nei fatti, il primo tentativo effettuato per realizzare la funzione di confronto, prevedeva di utilizzare al suo interno un array multidimensionale per la memorizzazione dei coefficienti della matrice di similarità. I coefficienti, poiché organizzati per loro natura in una struttura di tipo matriciale, dopo essere stati calcolati da un'apposita funzione scritta in PHP, venivano memorizzati in un campo di tipo array multidimensionale di una tabella, poiché questa sembrava la scelta apparentemente più ovvia. Il loro recupero, per renderli disponibili nella funzione di calcolo della distanza, avveniva tramite un'opportuna query che provvedeva a prenderli da questa tabella ed a metterli in una variabile, sempre di tipo array multidimensionale, dichiarata internamente alla funzione stessa. In questa funzione il calcolo della distanza, fatto ovviamente

mediante la formula indicata, avveniva quindi operando su due strutture di tipo array monodimensionali (gli istogrammi delle immagini) e su una di tipo array multidimensionale (i coefficienti della matrice di similarità). In queste modalità, tenendo presente il fatto che una matrice di similarità riferita ad un'istogramma di 122 posizioni presenta 14884 coefficienti, effettuare una query di confronto, su un database popolato da 25 immagini, richiedeva circa otto minuti per confrontare un'immagine con tutte le altre memorizzate, un tempo davvero inaccettabile considerando che 25 immagini sono un numero abbastanza limitato. Un prospetto relativo al contesto appena descritto, dove in sequenza sono riportate la dichiarazione della tabella utilizzata per memorizzare i coefficienti e la funzione di calcolo della distanza, è riportato a seguire.

```
CREATE TABLE pesi(
        id_pesi SERIAL PRIMARY KEY,
        matrice_pesi REAL[122][122] NOT NULL
);

CREATE OR REPLACE FUNCTION confronto(vettore_uno REAL[],
vettore_due REAL[]) RETURNS REAL AS $$
DECLARE
  matrice REAL[][];
  desc_uno REAL[];
  desc_due REAL[];
  sum REAL;
BEGIN
  desc_uno := vettore_uno;
  desc_due := vettore_due;
  sum :=0;
  SELECT INTO matrice matrice_pesi FROM pesi WHERE id_pesi=1;
  FOR i IN 1..122 LOOP
    FOR j IN 1..122 LOOP
      sum := sum + ((matrice[i][j])*((desc_uno[i])-(desc_due[i]))*
              *((desc_uno[j])-(desc_due[j])));
    END LOOP;
  END LOOP;
  RETURN sum;
END;
$$ LANGUAGE 'plpgsql' STABLE;
```

Prospetto 2.2 - La prima definizione della tabella dei pesi e della funzione “confronto”

L'esecuzione della seguente query, nelle condizioni ipotizzate, ha impiegato, in una prova di ricerca realmente effettuata, un tempo di esecuzione totale di 518,195.709 millisecondi.

```
SELECT t1.id_immagine AS img_di_riferimento, t2.id_immagine AS
        img_confrontata, confronto(t1.descrittore,t2.descrittore) AS
        distanza
FROM immagini t1, immagini t2
WHERE (t1.id_immagine=63) AND
       (confronto(t1.descrittore,t2.descrittore)<1)
ORDER BY (confronto(t1.descrittore,t2.descrittore)) ASC;
```

Prospetto 2.3 - La query utilizzata per la ricerca

A questo punto è stata una logica conseguenza interrogarsi su cosa determinasse un tempo di query così elevato. La risposta è stata individuata riflettendo sulla considerazione alla quale ci si riferiva prima: PL/pgSQL fornisce le prestazioni migliori scrivendo funzioni orientate il più possibile alla sua natura SQL e non alle comuni caratteristiche che può presentare un linguaggio di programmazione puro. In particolare, in questo caso, il motivo che determina un tempo così lungo per l'esecuzione della query, è stato individuato nel fatto che PL/pgSQL non è ottimizzato per trattare strutture di tipo matriciale dalle dimensioni così elevate. Il recupero in blocco dei coefficienti e la loro computazione, utilizzando una struttura di tipo array multidimensionale per espletare queste operazioni, sono i principali fattori che rendono l'esecuzione della funzione estremamente pesante. Per ovviare a questa situazione è stato necessario ridefinire la tabella dei pesi in altro modo, con l'unico scopo di poter introdurre un metodo che rendesse i tempi di recupero e computazione dei coefficienti accettabili, ovvero mettersi nelle condizioni di definire e utilizzare un cursore.

La funzione PHP alla quale facevamo riferimento prima, calcola i coefficienti della matrice di similarità in ordine di riga. Consideriamo adesso la definizione della tabella dei pesi riportata nel paragrafo 2.3.3: da essa possiamo notare che ogni singola tupla della tabella rappresenta un peso e la sua chiave primaria. Questi pesi

vengono inseriti nella tabella nell'ordine esatto con il quale sono calcolati dalla funzione PHP: le prime 122 tuple contengono quindi i coefficienti della prima riga della matrice di similarità, le seconde 122 conterranno invece i coefficienti della seconda riga della matrice e così via fino alle ultime 122 tuple, che saranno relative ai coefficienti dell'ultima riga della matrice. Questa memorizzazione sequenziale, unita al fatto che i coefficienti della matrice di similarità sono sempre impiegati complessivamente nel calcolo della distanza, rendono possibile il loro recupero e il loro utilizzo nella funzione distanza mediante l'uso di un cursore. Quello che andiamo ad illustrare di seguito non sarebbe stato possibile, se la funzione per il calcolo della distanza avesse fatto un uso parziale dei coefficienti, identificandoli singolarmente in base alle loro coordinate di riga e colonna. Vediamo il perché di questa impossibilità, spiegando prima cosa è un cursore.

Un cursore può essere visto come una variabile che contiene il "result set" di una query. Un cursore viene sempre dichiarato unitamente ad una dichiarazione di query. Processare il "result set" di una query mediante l'uso di un cursore è un processo simile a quello che si fa processando il "result set" tramite un costrutto iterativo; ovviamente l'uso di un cursore prevede alcuni vantaggi. Non è nostro interesse illustrare in questa sede tutte le caratteristiche di un cursore ed in che modo queste caratteristiche possono aumentare le performance di certe elaborazioni; l'unico aspetto che ci interessa è quello relativo al fatto che, per un recupero sequenziale dei dati da una tabella, come quello che dobbiamo fare noi per disporre dei coefficienti all'interno della funzione, utilizzare un cursore è una soluzione molto più performante rispetto ad una SELECT INTO o a qualsiasi altra via che preveda la dichiarazione di una query internamente ad un LOOP.

Un cursore, dopo essere stato opportunamente dichiarato, attraversa tre distinte fasi: OPEN, FETCH e CLOSE. La prima e l'ultima sono subito comprensibili, aprono e chiudono il cursore. Nella seconda avviene invece il recupero vero e proprio dei valori: ogni singola istanza della fase di FETCH recupera una tupla appartenente al "result set" della query associata al cursore. L'introduzione di un cursore ha quindi permesso di ridefinire la funzione distanza (riportata nel prospetto 2.4 alla pagina

seguente) in modo più performante.

La solita query di confronto, cioè esattamente quella riportata nel prospetto 2.3, se viene eseguita dopo aver ridefinito la funzione come riportato a seguire, per confrontare un'immagine con tutte quelle archiviate in un database popolato con 100 immagini, impiega meno di 20 secondi.

```
CREATE OR REPLACE FUNCTION
confronto(vettore_uno NUMERIC[],  vettore_due NUMERIC[],
dimensione INTEGER) RETURNS NUMERIC AS $$
DECLARE
 desc NUMERIC[];
 sum NUMERIC;
 pesi_curs CURSOR FOR SELECT peso FROM pesi ORDER BY id_peso;
 matval NUMERIC;
BEGIN
  FOR i IN 1..dimensione LOOP
    desc[i] := (vettore_uno[i]-vettore_due[i]);
  END LOOP;
  sum :=0;
  OPEN pesi_curs;
  FOR i IN 1..dimensione LOOP
    FOR j IN 1..dimensione LOOP
      FETCH pesi_curs INTO matval;
      sum := sum + (desc[i]*desc[j]*matval);
    END LOOP;
  END LOOP;
  CLOSE pesi_curs;
  RETURN sum;
END;
$$ LANGUAGE 'plpgsql' STABLE;
```

Prospetto 2.4 - L'implementazione finale della funzione “confronto”

I risultati parlano da soli: il singolo recupero dei coefficienti tramite l'uso di un cursore ed il loro impiego all'interno della funzione, svincolato dal fatto di doverli riferire tramite gli indici necessari quando questi sono posti in un array multidimensionale, aumenta le prestazioni in maniera davvero considerevole. Possiamo inoltre notare che nella nuova funzione è presente un altro parametro in ingresso: “dimensione”. Tramite questo parametro viene inoltrata alla funzione la dimensione degli istogrammi che devono essere confrontati. L'introduzione di questo parametro non è volta a migliorare le prestazioni, serve solo per rendere riutilizzabile la funzione, qualora il numero di colori dell'istogramma dovesse variare, possibilità che, come abbiamo detto nel capitolo precedente, l'applicazione offre modificando dal file di configurazione (setting.php) le componenti HSI dalle quali vengono generati i colori della mappa. Dovrebbe essere evidente, osservando il codice della funzione di confronto, che se nel calcolo della distanza, i coefficienti della matrice di similarità non venissero impiegati in sequenza, conoscendo a priori in che ordine sono stati memorizzati nella tabella che li mantiene, questo tipo di implementazione non sarebbe possibile. La stessa cosa accadrebbe se la funzione distanza facesse uso solo di determinati coefficienti, dovendo così identificare ogni singolo coefficiente utilizzato tramite le coordinate di riga e colonna della matrice di similarità.

Facciamo un'ultima considerazione, prima di terminare questo capitolo. Siamo arrivati a definire una funzione di confronto tra istogrammi e possiamo quindi realizzare l'applicazione web, con la quale archiviare e ricercare le immagini per contenuto cromatico utilizzando un database. Benché lo scopo sia stato raggiunto, può essere comunque lecito chiedersi, alla luce di quanto è emerso implementando la funzione di confronto in PL/pgSQL, se esistono anche altre strade per realizzare una funzione di questo tipo che risulti ancora più performante di quella appena descritta. A questa domanda non posso dare una risposta dimostrabile in pratica, poiché, oltre alla realizzazione in PL/pgSQL, non ne ho provate altre, però posso comunque fornire delle indicazioni di massima in riguardo. Conducendo delle indagini durante il primo approccio all'implementazione della funzione, è stata

abbastanza netta la sensazione che dovendo lavorare con strutture di tipo array, sia mono, che multidimensionali, sarei andato incontro, come poi è puntualmente successo, ad una situazione nella quale avrei dovuto adattare queste strutture al modo di operare di PL/pgSQL. Per evitare questo adattamento, posso affermare con certezza quasi assoluta, sarebbe bastato scrivere la funzione in C, linguaggio di programmazione che non presenta alcun problema quando deve trattare dati a struttura vettoriale o matriciale. Così facendo, sarebbero però sorti altri problemi. Anche se PostgreSQL permette di estendere le funzionalità del server mediante la scrittura di funzioni C, questo non è nei fatti un linguaggio procedurale, ma viene considerato un linguaggio esterno. PostgreSQL identifica un linguaggio procedurale server-side a seconda del fatto se questo può essere installato e disinstallato dalla macchina. Nel caso del C, la sua disinstallazione non è possibile, poiché questo è presente di default in qualsiasi distribuzione di PostgreSQL e non può essere rimosso. Questo fatto, seppur a mio modo di vedere risulti abbastanza strano, non è poi la fonte principale dei problemi ai quali facevo prima riferimento, benché in qualche modo vi influisca. Infatti, quando viene creata un'estensione del server, tramite una funzione scritta in un linguaggio procedurale, il codice sorgente ed il codice oggetto della funzione sono memorizzati in tabelle interne al database. Quando invece l'estensione al server è creata definendo una funzione in linguaggio esterno, come paradossalmente è considerato il C, nonostante questo sia distribuito con PostgreSQL e non sia da esso rimovibile, il codice della funzione non viene memorizzato nel database. Viene invece memorizzato in una libreria condivisa e linkato dal server al primo utilizzo. Per essere più precisi, in ambiente Windows, questo significa compilare una funzione C come file .dll (Dynamic Link Library) e rendere disponibile questo file al server di PostgreSQL. Siccome PostgreSQL nasce in ambiente Unix/Linux, dove l'operazione equivalente a quella appena descritta è quella di compilare la funzione C come file .so (Shared Object), questo processo effettuato in ambiente Windows non sembra ancora poter dare le dovute garanzie di funzionamento. Infatti, solo dalla versione 8.0, PostgreSQL server è stato portato in ambiente Windows e riconosciuto come applicazione nativa dello stesso sistema

operativo, senza richiedere la preventiva installazione dell'infrastruttura Unix-like Cygwin, prima necessaria per il corretto funzionamento di PostgreSQL in ambiente Microsoft Windows.

Ritenendo conclusa la trattazione relativa all'implementazione della funzione di confronto, possiamo passare all'argomento successivo, ovvero alla descrizione di come potrebbe essere velocizzata una ricerca condotta tramite l'impiego della funzione appena realizzata. A tale scopo è necessario introdurre un'altra caratteristica di PostgreSQL: gli indici.

2.5 - La possibilità di creare un indice

Quello che ci apprestiamo ad affrontare è un aspetto molto interessante, che potrebbe essere visto come il naturale punto di partenza per la progettazione di una seconda versione dell'applicazione. L'idea è quella di creare una struttura di supporto che permetta di avere dei tempi di elaborazione inferiori a quelli che possono essere ottenuti nelle condizioni attuali. Avremo modo di constatare, dalle prove fatte e dall'esame delle modalità con le quali l'applicazione riesce a produrre i risultati ottenuti, un fatto palese: i tempi di ricerca, quando questa viene effettuata nella modalità “Confronta con tutte le immagini” descritta nel paragrafo 3.3.1, crescono in maniera direttamente proporzionale all'aumentare del numero delle immagini contenute nell'archivio. Con gli strumenti offerti da PostgreSQL ed utilizzati fino a questo punto per poter effettuare le ricerche, non è possibile abbassare ulteriormente i tempi di elaborazione impiegati dall'ORDBMS per effettuare i confronti tra le immagini. PostgreSQL, come spero sia stato compreso dalla trattazione fatta in tutto questo capitolo, è però un sistema di gestione di basi di dati che per natura offre grandi possibilità di estensione delle proprie funzionalità. Questo significa che anche se al momento non è disponibile una struttura preconfezionata adatta a questa esigenza, è sempre ipotizzabile cercare di individuare se esiste un metodo per mettere PostgreSQL nelle condizioni di aumentare le performance dell'applicazione. Possiamo anticipare che questa

possibilità esiste, ma prima di esporla è necessario introdurre una caratteristica di PostgreSQL fino ad ora mai messa in evidenza: la creazione degli indici. Un indice è una struttura che sostanzialmente, almeno per quanto riguarda PostgreSQL, serve principalmente a fare due cose: garantire l'unicità dei dati e soprattutto velocizzare l'accesso ad essi.

Uno dei vincoli più importanti, quando si definisce una tabella, è rappresentato dalla dichiarazione della chiave primaria (PRIMARY KEY); ricordiamo che la chiave primaria può essere composta da uno o più campi della tabella. Quando PostgreSQL incontra una dichiarazione di tipo PRIMARY KEY, esso procede in maniera implicita alla creazione di un indice per quella chiave. In realtà, la solita cosa avviene anche dichiarando un campo, o un insieme di campi, di tipo UNIQUE: viene sempre creato un indice riferito ai campi interessati dalla dichiarazione. Appreso questo fatto sappiamo quindi che ognuna delle tre tabelle del database utilizzato da Image-Compare ha un indice definito sulla chiave primaria.

Abbiamo detto che uno degli utilizzi di un indice è quello di garantire l'unicità; abbiamo anche detto che gli indici vengono implicitamente creati quando viene dichiarato un vincolo che prevede l'unicità dell'insieme dei campi sottoposti a tale dichiarazione: sappiamo infatti che i campi sottoposti ai vincoli PRIMARY KEY e UNIQUE devono garantire di contenere dati unici all'interno di una tabella. Analizziamo quindi come un indice sia di aiuto per perseguire questo scopo, ovvero come consenta al DBMS di garantire l'unicità di determinati campi sottoposti a vincolo. Quando viene inserito un valore in un campo di una tabella, e questo valore deve essere unico, il DBMS deve controllare che non esista una tupla che in quel campo contiene già quel valore. Se le tuple della tabella sono ordinate in maniera del tutto casuale, il DBMS dovrà controllare tutte le istanze, per avere la sicurezza che il valore candidato all'inserimento non sia già presente. Se invece esiste una struttura che indicizza, secondo un qualche criterio, i valori contenuti nel campo dichiarato UNIQUE o PRIMARY KEY, sicuramente il DBMS avrà vita più facile nel verificare l'unicità dei valori.

Un indice, di fatto, rappresenta quindi una struttura che consente a PostgreSQL di

indicizzare i valori contenuti nel campo al quale l'indice si riferisce. Supponiamo di avere una relazione “persona”, all'interno della quale esiste un campo “codice_fiscale”, che rappresenta la chiave primaria naturale della tabella. Se per un qualunque motivo vogliamo che non esistano degli omonimi, dobbiamo dichiarare UNIQUE gli ipotetici campi “nome” e “cognome” della tabella. A questo punto, come già detto, PostgreSQL procede alla creazione di un indice per questi due campi ed in fase d'inserimento sfrutterà questo indice per controllare se la nuova persona, che vuole essere inserita nel database, non sia già presente. In assenza di un indice, il DBMS avrebbe dovuto scorrere tutte le tuple della tabella persona, per verificare se la nuova istanza rispetta il vincolo. In questo caso il criterio d'indicizzazione può essere quello alfabetico.

Appreso questo primo utilizzo di un indice, prima di proseguire, può essere utile chiarire un concetto, ovvero quale è la differenza tra un “clustered index” e un “non-clustered index”. Un indice che opera nelle modalità descritte dall'esempio è sempre realizzato tramite l'impiego di una struttura ad albero, utilizzata in questo caso per ridurre il tempo necessario a verificare l'unicità dei dati. Un “clustered index” prevede che i nodi foglia dell'albero siano di fatto i valori delle tuple della tabella. In un “non-clusterd index”, invece, i nodi foglia contengono i puntatori alle tuple della tabella; tali tuple non sono quindi mantenute effettivamente in ordine, benché tramite i puntatori sia comunque possibile indicizzarle secondo un certo criterio. Un “clustered index” risulta quindi molto efficiente quando viene fatto un accesso sequenziale ai dati, ma risulta anche molto costoso da mantenere rispetto ad un “non-clustered index”. PostgreSQL non supporta gli indici di tipo “clustered”.

Discutendo della differenza tra gli indici di tipo “clustered” e “non-clustered”, abbiamo parlato di accesso ai dati. Come sarà oramai evidente e come era stato preannunciato, un indice è una struttura che, nelle modalità in cui consente di verificare più rapidamente l'unicità dei dati, permette anche di effettuare un accesso più veloce ai dati stessi. Risulta ovvio che in seguito ad una SELECT, dove viene espressa una certa condizione nella clausola WHERE, recuperare i dati sarà più semplice se questi sono indicizzati. Sempre nelle ipotesi della relazione “persona”,

supponiamo di voler recuperare i dati di tutti gli individui i cui cognomi iniziano con le lettere che vanno dalla 'A' alla 'F'. Questo è possibile tramite una query di questo tipo:

```
SELECT * FROM persona WHERE (cognome>='A') AND (cognome<='F');
```

Se non esiste un indice definito per il campo “cognome”, è evidente che dovrò scandire tutte le tuple della tabella per determinare il dataset. Questa operazione prende il nome di '*full table scan*'. Se invece è presente un indice è possibile recuperare tutte le tuple d'interesse utilizzando la struttura ad albero mediante la quale è stato indicizzato il campo. Come già detto, in questo caso di accesso, un “clustered index” è più performante rispetto ad “non-clustered index”. Indipendentemente dal tipo di indice, questa operazione è chiamata '*partial index scan*'. Inutile sottolineare che la presenza di un indice rende l'accesso ai dati molto più veloce.

PostgreSQL dispone di diversi tipi di indice: quello al quale abbiamo implicitamente fatto riferimento fino ad ora è l'indice “B-Tree” (Balanced Tree). Solitamente, di default, quando PostgreSQL crea un indice in maniera automatica o non riceve diversa indicazione, crea un indice di questo tipo. Un altro tipo di indice fornito da PostgreSQL è l'Hash Index. Questo indice opera indicizzando i dati in base al risultato di una funzione Hash e, in virtù di ciò, i suoi tempi di accesso sono tanto migliori quanto è migliore la funzione: minori saranno le collisioni determinate dalla funzione e minore sarà il tempo di accesso. Dobbiamo notare che questo indice, proprio perché utilizza come criterio di indicizzazione il risultato di una funzione Hash, non offre particolari vantaggi nel recupero di range di valori sequenziali, come ad esempio avviene nel caso della SELECT di inizio pagina. Un Hash Index è particolarmente adatto per recuperare valori singoli, valori basati sull'uguaglianza. Un esempio può essere la seguente query:

```
SELECT * FROM persone WHERE cognome='Pratesi';
```

PostgreSQL supporta altri due tipi di indice: R-Tree e GiST. Il primo è utilizzato per indicizzare dati di tipo spaziali, quali quelli geometrici e geografici, supportati da PostgreSQL. Il secondo è un indice di tipo B-Tree, che però può essere esteso con l'aggiunta di nuovi predicati per effettuare l'indicizzazione dei dati. Quest'ultima possibilità offerta dall'ORDBMS non è cosa da poco, infatti, la possibilità di estendere i predicati, permette anche di definire nuove metriche da utilizzare per indicizzare dati di tipo strutturato appartenenti ad un certo spazio. A questo punto dovrebbe iniziare ad illuminarsi la strada che ci può portare verso la meta, ma ancora dobbiamo far luce su alcune questioni fondamentali prima di poter ritenere concluso questo percorso.

Sappiamo bene che PostgreSQL supporta il tipo ARRAY e quindi è ovvia conseguenza la possibilità di definire un indice per i vari elementi di un campo dichiarato di tipo array. A questo punto, definendo normalmente un indice per questo campo, l'indice lavora nelle stesse modalità con le quali opera su un campo rappresentato da un dato non strutturato, supposto che il campo array sia dichiarato monodimensionale. La situazione cambia, invece, se vogliamo indicizzare l'intera struttura ARRAY, definendo per essa una certa metrica. Consideriamo con finalità puramente didattiche questo esempio, supponendo per ipotesi che non esistano altri modi già preconfezionati per raggiungere il medesimo scopo.

Prendiamo il caso in cui un campo “coordinate” di una tabella è dichiarato di tipo ARRAY, con dimensione pari a tre, volto a rappresentare le coordinate spaziali X,Y e Z di un generico punto nello spazio. Con i soli predicati utilizzabili da un indice B-Tree, non è possibile definire un indice per questo campo, che possa indicizzare le varie tuple di coordinate spaziali, in base alla loro distanza dal centro. Se invece posso introdurre un predicato che calcola il modulo di ogni vettore, posso indicizzare i vari punti, da quello più vicino a quello più lontano dall'origine del sistema di riferimento scelto, supposto in questo caso essere il centro degli assi X,Y e Z. A questo punto, anche un'ipotetica query che vuole individuare tutti i punti presenti nella tabella, contenuti in una sfera di raggio pari ad un dato valore, potrà essere velocizzata grazie al supporto fornito dall'indice.

L'esempio fatto non è forse il più adeguato per evidenziare la necessità di un'estensione, poiché, in tali circostanze, come detto, PostgreSQL dispone già dell'indice R-Tree, progettato appositamente per indicizzare i dati spaziali. Ciò non toglie che tale esempio dovrebbe essere stato comunque sufficientemente propedeutico per comprendere come PostgreSQL, oltre a permettere di estendere dati e operatori, consenta di creare applicazioni dalle prestazioni elevate anche quando operano su nuovi tipi di dati definiti dall'utente, proprio grazie alla possibilità di definire per essi indici appositamente costruiti.

Procediamo adesso ad un'ultima considerazione: un indice ha sicuramente il grande pregio di rendere le operazioni di ricerca molto veloci, però, per ottenere questo aumento di performance nella fase di recupero dei dati, c'è sempre e comunque un prezzo da pagare in fase d'inserimento, che può essere più o meno oneroso.

In riguardo a questo aspetto, abbiamo già fornito delle informazioni inerenti all'argomento parlando dei “clustered index” e dei “non-clustered index”. Dicendo infatti che il mantenimento di un “clustered index” è molto costoso, veniva sottinteso che mantenere l'ordinamento comporta dei costi extra per ogni nuova fase d'inserimento. Utilizzando un “non-clustered index” non si devono invece sopportare i costi dovuti al mantenimento delle tuple ordinate in sequenza e quindi l'inserimento è più veloce. In entrambi i casi, c'è comunque un costo fisso, ossia quello dovuto ai vari test necessari ad ogni inserimento per mantenere aggiornato l'indice. Questi test, quando il predicato di confronto è uno di quelli utilizzati da un indice B-Tree, mediamente non sono molto onerosi, poiché l'indicizzazione del nuovo valore generalmente non è molto complessa. La situazione cambia se l'indice è creato su dati strutturati, sui quali definisco una nuova metrica ed un predicato che possa permettere la loro indicizzazione. I costi di mantenimento dell'indice saranno dipendenti sia dal dato che dalla metrica su esso definita. Nelle situazioni in cui il predicato è rappresentato da una funzione computazionalmente pesante, l'inserimento sarà quindi rallentato in maniera direttamente proporzionale alla complessità della funzione.

Ripensiamo ora all'esempio fatto in precedenza, relativamente all'indicizzazione di

un punto spaziale in base alla sua distanza dal centro del sistema di riferimento; cosa potrebbe accadere se si volesse indicizzare ogni singolo punto in base alla sua distanza dagli altri punti dell'insieme? La risposta a questa domanda prevede una trattazione non affatto banale. In particolare, strutture, volte alla creazione di indici che possano operare in contesti di questo genere, sono il presupposto alla base dei problemi di "proximity search". Con questo termine vengono identificati i problemi di ricerca tramite query di oggetti in un dataset usando una certa metrica. Il fine è quello di preprocessare l'insieme per minimizzare il numero di valutazioni della distanza al momento della query. Una notevole differenza sussiste tra il caso in cui gli oggetti sono punti di uno spazio euclideo N-dimensionale, a quello in cui lo spazio è un generico spazio metrico, dove la distanza non è quella euclidea, ma una funzione che deve solo soddisfare la disuguaglianza triangolare. Nel primo caso esistono già soluzioni ben conosciute: l'implementazione della struttura alla base dell'indice R-Tree è fondata su una di queste. Negli altri casi invece la situazione diviene inevitabilmente più complessa. Esistono due strade concettuali da seguire, che citiamo a solo titolo informativo. La prima si basa sui diagrammi di Voronoi, mentre la seconda è fondata sulla trasformazione dello spazio metrico in uno spazio D-dimensionale e conduce alla definizione di una famiglia di algoritmi chiamati pivot-based.

Siamo quindi giunti alla fine di questo percorso: per creare una struttura di validità generale, volta ad indicizzare nel loro complesso i campi di tipo vettore, che in questo caso rappresentano gli istogrammi di colore di ogni singola immagine, e ridurre così il tempo di query, quando vengono effettuate delle ricerche tramite la funzione "confronto", o con qualsiasi altra funzione diversa da quella utilizzata ma comunque sempre adatta allo scopo, è necessario il supporto di un algoritmo di tipo pivot-based. Sempre a titolo informativo, diciamo che escludiamo la soluzione basata sui diagrammi di Voronoi, poiché, dalla fonti consultate per documentarsi su questi argomenti, traspare che questa soluzione fornisce buone prestazioni in spazi dimensionali di bassa dimensione, che non rappresentano quindi il caso d'interesse.