ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

TITOLO DELLA TESI

SVILUPPO DI UN'INTERFACCIA PER LA RICERCA ASSISTITA IN DATABASE DI IMMAGINI DI BIOLOGIA CELLULARE

TESI IN

RETI LOGICHE LA

Relatore
Prof. Alessandro Bevilacqua

Presentata da
Andrea Ghetti

Co-Relatori
Ing. Alessandro Gherardi

Sessione I
Anno Accademico 2011-2012

PAROLE CHIAVE

DATABASE APPLICATION
BIOLOGIA CELLULARE
IMAGE RETRIEVAL

Indice

Capitolo 1

Introduzione

La biologia (dal greco *bios*, vita e *logos*, studio) è la scienza che studia la vita. In particolare la biologia cellulare è una branca che studia l'unità base della vita, la cellula. Ha a che fare con tutti gli aspetti della cellula che includono l'anatomia, la divisione (mitosi e meiosi) e tutti i processi come la respirazione e la morte cellulare.

L'origine della biologia cellulare è strettamente legata allo sviluppo delle tecniche di osservazione che permettono di analizzare la cellula e in particolare allo sviluppo della microscopia. Si è infatti passati nel tempo dalle sperimentazioni *in vivo* dove si faceva uso di organismi nella loro integrità strutturale e fisica, alla sperimentazione *in vitro* dove l'esperimento è condotto usando una componente di un'organismo che è stato isolato dal suo abituale contesto biologico in modo da permettere una più conveniente e dettagliata analisi che si potrebbe avere invece con l'intero organismo. Il passaggio alla sperimentazione *in vitro* fa anche parte della naturale evoluzione scientifica che ha riguardato l'intera biologia durante il XIX secolo in modo da indirizzare l'interesse della comunità scientifica verso livelli di organizzazione (cellulare e molecolare) sempre più infinitesimale.

La storia delle colture cellulari è recente. Iniziò nel XIX secolo dove Schleiden e Schwann ipotizzarono che la cellula fosse un'unità funzionale capace di vita autonoma, tuttavia questo non portò subito a effetti pratici nella biologia sperimentale [16]. I primi esperimenti furono svolti nel 1885 da Wihlelm Roux che riuscì con sucesso ad espiantare e mantenere vivo per pochi giorni in una soluzione salina calda il cervello di un'embrione di pollo. Agli inizi del XX secolo iniziarono i primi studi sull'ambiente cellulare. I ricercatori tentarono di isolare cellule di organismi e di mantenere

in coltura frammenti di tessuti vitali. I primi esperimenti riguardarono soprattutto animali a sangue freddo (rane, crostacei) perchè questi tessuti non necessitavano di essere mantenuti a temperature diverse da quella ambientale. Una svolta decisiva nelle colture si ha nel 1943 dove Earle riuscì ad otterene le prime linee continue di cellule di un mammifero. L'uso delle colture cellulari subisce quindi un forte incremento agli inizi degli anni Cinquanta ma soltanto negli ultimi decenni è divenuta una pratica comune essendo ormai possibile mantere a lungo le cellule in coltura grazie al progresso delle scienze moderne e della tecnologia.

I moderni laboratori di ricerca sono composti da stanze a temperatura controllata, autoclavi, frigoriferi normali e a basse temperature, centrifughe, bilancie analitiche e microscopi biologici. I microscopi biologici sono caratterizzati di un'elevato potere di ingrandimento e sono usati per osservare preparati su vetrini contenenti organi o sezioni di tessuto. Possono avere una testa monoculare, binoculare o trinoculare per consentire l'installazione di una fotocamera.

(a) Microscopio monoculare

(b) Miscroscopio binoculare

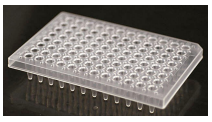
(c) Miscroscopio trinoculare

I principali parametri che caratterizzano un microscopio sono due: l'ingrandimento e la risoluzione [23]. L'ingrandimento è il valore che indica quanto più grande appare l'oggetto rispetto alle sue dimensioni reali. La chiarezza dell'immagine ingrandita dipende invece dal potere risolutivo o risoluzione che è la capacità dello strumento ottico di mostrare in modo nitido e distinto due oggetti separati tra loro. Il potere risolutivo di un microscopio ottico (*LM Light Microscope*) che utilizza la luce che attraversa il campione, è di circa $0{,}2\mu m$ che equivale alle dimensioni di una piccola cellula, limitandone l'ingrandimento utile a 1000X. I microscopi elettronici (*EM*

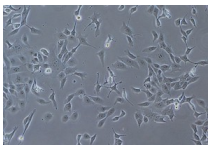
Election Microscope) a differenza di quelli ottici non utilizzano la luce ma un fascio di elettroni, hanno un potere risolutivo molto maggiore e i più potenti possono distinguere oggetti di soli 0.2nm, con un'ingrandimento di 100.000X col quale è possibile vedere all'interno della cellula. Un miglioramento di migliaia di volte rispetto a quello ottico. Nonostante questo maggior potere risolutivo i microscopi ottici, a differenza di quelli elettronici, non necessitano dell'uccisione della cellula per eseguire l'analisi del campione e per questo motivo sono ancora largamente utilizzati per l'analisi delle cellule vive.

Durante un'esperimento i ricercatori analizzano i pozzetti (*well*) contenenti i tessuti ed eventuali colture. Di questi pozzetti vengono prese diverse immagini periodizzate nel corso dell'esperimento, durante la fase di accrescimento cellulare, acquisite attraverso uno strumento di *imaging*, queste immagini insieme alle annotazioni prese devono venire catalogate con la possibilità di poter essere recuperate in modo semplice e funzionale.

(d) Pozzetto singolo



(e) Piastra di pozzetti



(f) Immagine ricava da un pozzetto

Soluzione a questo problema potrebbe essere l'utilizzo di un *software* facente uso di un *database*(db) e metadati per la registrazione delle informazioni [26]. La base dati risulterebbe utile per la memorizzazione delle informazioni mentre i metadati (dal greco *meta*, oltre e dal latino *datum* infomazione) descivono la struttura dei dati, la restrizione sui valori ammissibili dei dati, le relazioni tra di essi. I metadati quindi consentono di ottimizzare le azioni di:

- Ricerca: la possibilità di ricercare le informazioni e dati da qualsiasi oggetto memorizzato.
- Localizzazione esatta: la possibilità di identificare e localizzare rapidamente una particolare informazione nell'oggetto.
- Selezionare ed estrapolare: la possibilità di poter selezionare ed estrapolare i dati rapidamente in funzione di particolare ricerche filtrate.

- Interazione semantica: la possibilità di poter effettuare ricerche in settori disciplinari diversi in funzione di una serie di relazioni tra i descrittori.
- Management dei *database*: la possibilità di poter gestire dei *database* e di farli interagire su piattaforme diverse.
- Fruizione continua e immediata: la possibilità di accedere ad ogni istante ai dati e di estrapolarli a proprio piacimento.

Si è pensato di poter utilizzare i metadati per l'inserimento delle proprietà nelle immagini. Grazie a questi sarà possibile in futuro, utilizzando un programma di elaborazione delle immagini interfacciato al *database* l'inserimento in queste proprietà dinamiche di *features* legate alla proprietà semantica dell'immagine. Questo consentirà in futuro al biologo di poter effettuare, attraverso determinati algoritmi di estrazione, *query* per immagini [13]. Attraverso l'implementazione di tale software si potrebbe quindi facilitare e migliorare le fasi della ricerca bio-cellulare nei laboratori.

Molti produttori di microscopi e/o telecamere per l'acquisizione di immagini da essi, offrono prodotti di questo tipo. Questi programmi però, oltre ad essere molto costosi, in media circa mille euro per una licenza, spesso sono utilizzabili solamente con il microscopio o la fotocamera del produttore. Inoltre non contengono tutti le funzionalità richieste dal gruppo di lavoro portando quindi all'utilizzo di ulteriori programmi i quali risultano molto difficili da utilizzare dal personale di laboratorio perchè fanno uso di una logica e un'interfaccia poco comprensibile; allo stesso tempo l'utilizzo di questi *software* risulta molto importante per la gestione di un laboratorio di ricerca migliorandone sotto tutti gli aspetti le fasi della ricerca bio-cellulare.

Questo lavoro di Tesi è mirato allo sviluppo ed implementazione di un programma per la gestione e ricerca delle immagini ricavate da esperimenti di biologia cellulare. A differenza dei prodotti in commercio si pone come obiettivo la creazione di un software implementato secondo il punto di vista di chi esegue gli esperimenti e secondo uno standard definito dal gruppo di utilizzo, altra caratteristica fondamentale dovrà essere la facilità d'uso e l'eliminazione delle limitazioni che quelli attualmente in commercio impongono, ad esempio l'uso indifferente di qualsiasi marca per l'acquisizione delle immagini, la possibilità di poter essere utilizzato su diverse piattaforme (Windows, OSX, Linux..), il tutto attraverso un'interfaccia *user friendly*. Al tempo stesso il programma deve offrire funzioni di catalogazione e ricerca

semplici ma efficaci. In particolare in questo lavoro di Tesi viene descritto il percorso effettuato per creare la parte di catalogazione dell'applicazione partendo da una bozza di programma preesistente, modificandola e migliorandola per poi passare alla parte di ricerca utilizzando le proprietà dei metadati.

Questo elaborato di Tesi è composto dai seguenti capitoli:

- Capitolo 2: descrive nei dettagli il percorso effettuato per la scelta del *database engine* dell'applicazione, dalla descrizione dei vari DB presi in considerazione, ai test effettuati per eseguire la comparazione fino alla scelta finale.
- Capitolo 3: riporta i passi effettuati nell'analisi e implementazione della parte di catalogazione delle immagini, dal riuso del codice esistente all'implementazione di nuove funzioni.
- Capitolo 4: descrive nei dettagli il percorso effettuato per la realizzazione della parte di ricerca delle immagini, dall'analisi delle esigenze, alla generazione del *layout* di ricerca fino all'implementazione.
- Capitolo 5: mostra e descrive le funzioni e le cartatteristiche dell'applicazione realizzata.

Capitolo 2

Scelta del DB engine

Questo lavoro di Tesi descrive la prima parte di un progetto che porterà ad avere un programma per la catalogazione e ricerca semantica delle immagini raccolte da esperimenti di biologia cellulare. La scelta del *database* in un'applicazione è fondamentale, questo deve rispettare le proprietà che la nostra applicazione deve avere, se si vuole, ad esempio, sviluppare un'applicazione *real-time* si avrà bisogno di un db che esegua delle transizioni molto rapide, cosa che non sarà necessaria in un'applicazione che dovrà gestire il magazzino di un'azienda. La scelta del giusto *database* è quindi determinante per l'applicazione e condizionerà poi anche le future scelte sulla creazione e implementazione della ricerca delle immagini.

Possiamo suddividere i *database* in base al relativo modello logico di cui fanno uso. Si definisce come modello logico la collezione di strutture che rappresentano la base dati. Questa è indipendente dalla piattaforma *hardware* o dal sistema operativo ma dipende dal tipo di *DBMS (Database Management Sytem)* sul quale dovrà funzionare. Il modello relazionale è il più diffuso, qui lo schema viene visto, in prima approssimazione, come una collezione di tabelle e relazioni tra esse [4]. Modelli utilizzati in passato ma oggi poco diffusi sono quello gerarchico dove le informazione sono strutturate secondo uno schema ad albero e quello reticolare dove il *database* è modellato come un reticolo e la struttura è diversa rispetto a quello ad albero solamente per la possibilità di ogni nodo d'avere più di un nodo padre [20]. Negli ultimi anni si stanno diffonendendo le basi dati basate sul modello logico orientato agli oggetti dove i dati vengono strutturati in base al paradigma orientato agli oggetti [21]. Lo schema sarà quindi composto da oggetti contenenti classi, attributi e aggregazioni con altre classi. Usando questo tipo di *database* la

memorizzazione degli oggetti avviene senza dover prevedere un *mapping* tra questi e le tabelle.

2.1 Criteri di scelta

Il *database* deve ripecchiare le caratteristiche dell'applicazione, essendo questa sviluppata nel linguaggio C#, la base dati scelta dovrà essere capace di interfacciarsi con esso.

Altra caratteristica presa in esame è stata la portabilità, nel caso l'applicazione venisse prodotta anche per altri sistemi operativi oltre a Windows, questo garantirebbe una riduzione del tempo di sviluppo e dei costi [2]. La portabilità dei programmi dipende essezialmente dal linguaggio di programmazione utilizzato. Certi linguaggi non sono portabili perchè per alcuni ambienti non esiste un compilatore o un interprete o perchè questi compilatori o interpresti presentano delle modifiche più o meno rilevanti al codice che accettano o alla semantica che attribuiscono ad alcuni costrutti. Questo problema può essere risolto utilizzando uno standard del linguaggio per i quali vi sono compilatori ed interpreti certificati. Un'esempio è il linguaggio C di cui è stato formito prima uno standard ANSI[1] e poi ISO[2]. I programmi scritti in C sono quindi portabili nel senso che possono essere compilati in un qualsiasi ambiente. Altra forma di portabilità viene fornita dai linguaggi interpretati. Un'esempio in questo caso è *Java*, dove il *software* può essere eseguito su qualunque ambiente che abbia installata la *Java Virtual Machine*.

Nel nostro caso il linguaggio C#, sviluppato da Microsoft, è stato approvato come standard *ECMA (European Computer Manufacteurers Association)* nel 2001 col rilascio di ECMA-334 C# Language Specification [9] ed è diventato standard ISO nel 2003. Secondo lo standard redatto da ECMA è stato sviluppato il progetto *open source* Mono il quale comprende strumenti come il compilatore per C#. Mono funziona sui sistemi Linux, Mac OSX e Windows. Grazie a questo progetto quindi il *software* sviluppato sarà portatile.

Nel caso di applicazione commerciali, anche il tipo di licenza con cui viene rilasciato il db engine deve essere preso in considerazione. Con l'espansione

[1] ANSI o *American National Standards Institute* è un'organizzazione privata senza fini di lucro che produce standard industriali per gli stati uniti. É un membro dell'ISO.

[2] ISO o *International Organization for Standardization* è la più importante organizzazione mondiale per la definizione di norme tecniche.

del *software* libero possiamo dividere in due grandi categorie i tipi di licenza: licenze per *software* proprietario e licenza per *software* libero ed *open source*. Nel primo caso ne fanno parte l'EULA *(End User License Agreement)*, shareware e freeware. Nel secondo caso ne sono presenti circa un centinaio ognuna delle quali con le proprie caratteristiche.

Essendo l'applicazione rivolta alla catalogazione e ricerca di immagini, la base dati scelta dovrà avere una buona velocità di inserimento record ed esecuzione di query. Infine, essendo stata sviluppata già in parte, anche la possibilità di poter effettuare il porting dal Microsoft SQL CE usato al db scelto è stata analizzata.

Riassumendo, i criteri per la scelta finale del *database* presi in considerazione sono stati:

- Interfacciamento col linguaggio C#
- Portabilità
- Tipo di licenza
- Prestazioni
- Facilità di porting

Caratteristiche dei database presi in esame

Microsoft SQL CE

Microsoft SQL CE è un *database* relazionale prodotto dalla Microsoft ed era quello usato nella parte di applicazione preesistente. Il motore di SQL CE è una versione minore di SQL Server, risulta molto compatto ed è pensato per la gestione di dati locali su *desktop* e *mobile*. Essendo fornito con un *provider* per .NET, è pienamente supportato il linguaggio C#, ma il suo uso è limitato alle piattaforme Windows e per poterlo inserire in un prodotto con finalità commerciali ha bisogno dell'acquisto di una licenza.

SharpSQL

SharpSQL è un progetto *open source*, distribuito sotto licenza GNU(GPL)[3], scritto interamente in C# e disponibile per più piattaforme. Il *database* è locale ed stato testato anche se non è stata rilasciata alcuna versione stabile [25].

Firebird

Firebird è un *database* relazionale *opensource*, distribuito sotto licenza IPL o IDPL[4]. Supporta molte piattaforme e numerosi linguaggi di programmazione, altra caratteristica è l'alto livello di conformita con gli stardard SQL [24].

MySQL

MySQL è un *database* relazionale multipiattaforma e possiede l'interfaccia per diversi linguaggi tra cui C#. Supporta la maggior parte della sintassi SQL e per prodotti di tipo commerciali ha bisogno dell'acquisto di una licenza altrimenti viene rilasciato sotto licenza GNU GPL [7].

Postgres SQL

Postgres SQL è un *database* relazionale ad oggetti, disponibile per più piattaforme, possiede l'interfaccia con C#. Viene distribuito con licenza libera PostgreSQL license, simile alla BSD[5], e usa il linguaggio SQL [18]. Solitamente viene utilizzato per progetti molto grandi e può essere un sostituto di prodotti a pagamento come Oracle, DB2 o liberi come Firebird e MySQL.

[3]La GNU *General Public License*, indicata con GNU GPL o semplicemente con GPL, è una licenza per *software* libero edita dalla Free Software Foundation Inc.

[4]La IPL *Interbase Public License* è una licenza di software libero simile alla *Mozilla Public License*

[5]Sono una famiglia di licenze permissive per software, molte sono considerate libere ed *open source*.

DB2 Express C

DB2 Express C è un *database* relazionale prodotto da IBM, versione gratuita del DB2 per uso non commerciale, utilizza il linguaggio SQL, scritto in C e C++, interfaccia pienamente il linguaggio C# ed è multipiattaforma [1].

INGRES

INGRES è un *database* relazionale *opensource* distribuito con licenza GNU(GPL), predisposto a supportare grandi applicazioni. Scritto in C, supporta il linguaggio SQL, si intefaccia col linguaggio C# ed è multipiattaforma. Per uso non commerciale è distribuito con licenza GNU GPL, altrimenti è necessario l'acquisto di una licenza [6].

SQLite

SQLite è un *database* relazionale di pubblico dominio[6] scritto in C ed utilizzato per applicazioni sia desktop che mobile. Sono presenti interfaccie per poterlo usare con diversi linguaggi e supporta SQL [11].

DB4Object

DB4O è un *database* ad oggetti *opensource* e per prodotti commerciali ha bisogno dell'acquisto di una licenza altrimenti viene distributito sotto licenza GNU GPL. Interamente scritto in Java e .NET si interfaccia con C# ed è multipiattaforma [3].

[6]Con l'espressione di pubblico dominio si indica in generale il complesso e la globalità dei beni insuscettibili di appropiazione esclusiva da parte di alcun soggetto pubblico o privato.

2.2 Comparazione

Per compararne le prestazioni è stato creato un test che consiste nell'eseguire su ciascun *database* quattro prove: una di *Insert*, una di *Update*, una di *Select* ed infine una di *Delete*. Ogni prova, ripetuta dieci volte, consiste nell'esecuzione di duecento *request* e se ne è cronometrato il tempo di secuzione. Da ciascuna prova sono stati tolti il miglior risultato ed il peggiore e delle restanti otto ne è stata fatta la media.

Per il test si è cercato di adoperare uno schema del *database* che potesse essere il più possibile similare a quello che poi avremmo utilizzato. Si è deciso di utilizzare tre tabelle in relazione tra loro secondo il seguente schema:

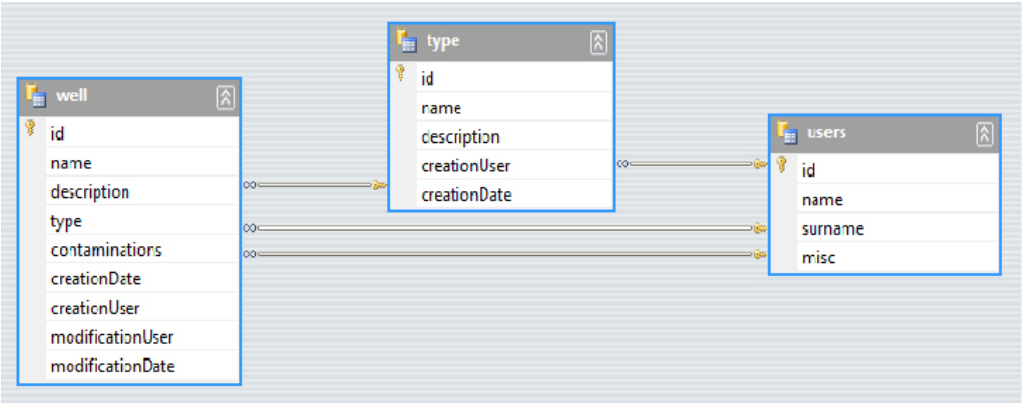


Figura 2.1: Schema *database* utilizzato per il test prestazionale.

In base ai risultati ottenuti i db sono stati suddivisi in tre gruppi secondo i tempi di risposta alla singola *request*.

Test Insert

Per questa prova sono stati inseriti prima mille record nelle tabelle *users* e *type* cronometrando il tempo per l'inserimento dei record nella tabella *well*. E' stata usata la seguente istruzione SQL:

```
INSERT INTO well
(id, name, description, type, contamination, creationDate,
creationUser, modificationUser, modificationDate)
VALUES
({0},{1},{2},{3},{4},{5},{6},{7},{8})
```

Eseguendo *query* di questo tipo, un problema che poteva nascere e invalidare i risultati era quello che la risposta venisse non dall'esecuzione della richiesta e quindi la ricerca su db ma dalla cache.

Molti *database* possiedono un sistema di *caching* interno che ne ottimizza il rendimento [10]. Questa caratteristica non fa altro che analizzare la *query* inviata e controllorne l'esistenza in *record* precedentemente estratti per evitare di rieseguirla. In pratica i risultati estratti, vengono salvati in memoria RAM e resi disponibili in caso di una richiesta dal *client*. All'arrivo di una richiesta identica ad una precedentemente presente in *cache*, per la risposta non sarà più neccessario accedere all'*hard disk* in lettura e applicare il filtro presente in *WHERE* ma verrà riportata la risposta presente precendentemente salvata generando un incremento delle prestazioni. Per evitare che ciò accada è necessario modificare anche minimamente la *query* da eseguire, infatti la *cache* si attiva se le richieste passate sono identiche, anche una maiuscola al posto di una minuscola non la attiverrano. Per evitare quindi problemi di *caching* durante l'esecuzione delle *query*, venivano automaticamente cambiati i valori di *creationUser*, *modificationUser* e di *type*.

I risultati sono stati raccolti nel seguente grafico:

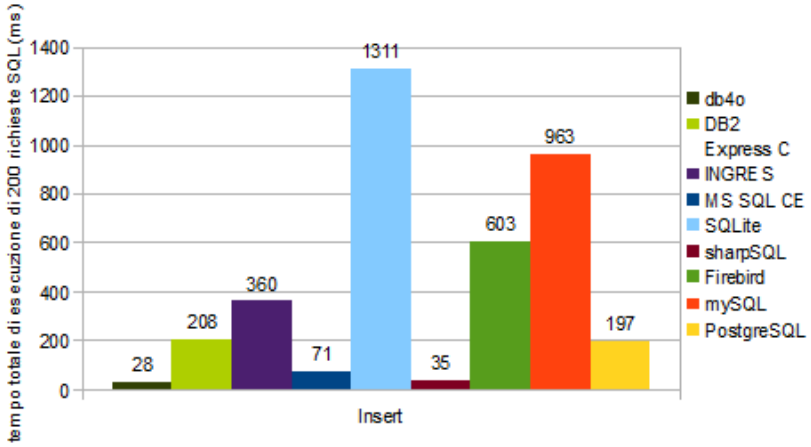


Figura 2.2: Grafico risultati del test di Insert

I *database* sono stati divisi in tre gruppi e raccolti in questa tabella in base al tempo di riposta ad una singola *request*:

$< 0{,}5$ ms		< 2 ms		> 2 ms	
DB4O	0,14 ms	Postgres SQL	1 ms	Firebird	3 ms
sharpSQL	0,18 ms	DB2 Exp. C	1 ms	MySQL	4,8 ms
MS SQL CE	0,36 ms	INGRES	1,8 ms	SQLite	6,6 ms

Test Update

Per questa prova sono stati mantenuti i mille record nelle tabelle users e type e portati a duemila i record della tabella well. Per la prova è stata utilizzata la seguente istruzione SQL:

```
UPDATE well
SET name = {0}, description = {1}, modificationUser = {2},
modificationDate = {3}
WHERE
creationUser = {4}
```

Per evitare i problemi di *caching* durante le *request* venivano modificati i valori di *creationUser*.

I risultati sono stati raccolti nel seguente grafico:

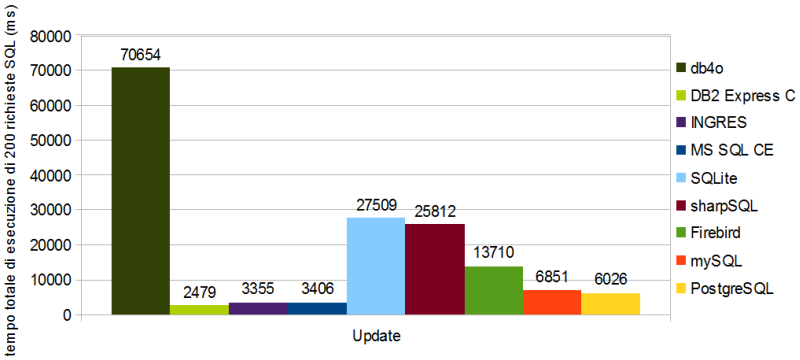


Figura 2.3: Grafico risultati del test di Update

I *database* sono stati divisi in tre gruppi e raccolti in questa tabella in base al tempo di riposta ad una singola *request*:

< 50 ms		< 150 ms		> 150 ms	
DB2 Exp.C	12,4 ms	Firebird	68,6 ms	DB4O	353,3 ms
INGRES	16 ms	sharpSQL	129,1 ms		
MS SQL CE	17 ms	SQLite	137,5 ms		
Postgres SQL	30,1 ms				
MySQL	34,4 ms				

Test Delete

Per effettuare questa prova sono stati aumentati i record nella tabella well a ventimila e sono stati cancellati alcuni di questi utilizzando la seguente istruzione SQL:

```
DELETE FROM well
WHERE modificationUser = {0}
```

In questa prova per evitare i problemi di *caching* veniva modificatio il valore di *modificationUser*.

I risultati sono stati raccolti nel seguente grafico:

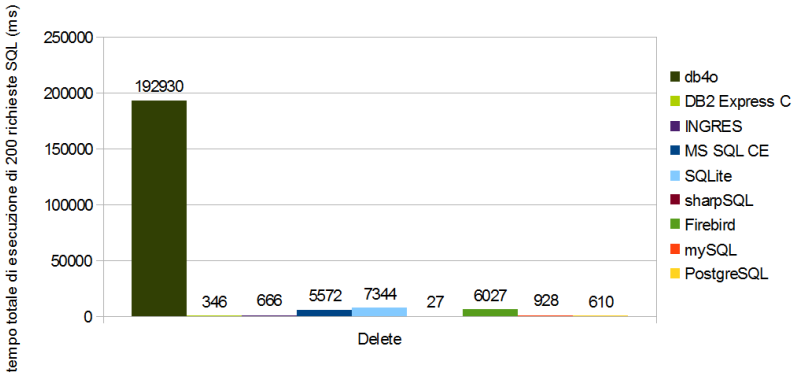


Figura 2.4: Grafico risultati del test di Delete

I *database* sono stati divisi in tre gruppi e raccolti in questa tabella in base al tempo di riposta ad una singola *request*:

< 10 ms		< 50 ms		> 50 ms	
sharpSQL	0,1 ms	MS SQL CE	27,9 ms	DB4O	964,7 ms
DB2 Exp. C	1,7 ms	Firebird	30,1 ms		
Postgres SQLE	3,1 ms	SQLite	37,7 ms		
INGRES	3,3 ms				
MySQL	5,4 ms				

In questa prova si è voluto lavorare con un numero cospicuo di record per testare anche l'affidabilità dei *database*. Solamente con sharpSQL si ha avuto un'aumento inspiegabilmente elevato delle dimensioni e dei tempi di esecuzione. Si è poi provveduto a cancellare l'istanza presente e rieseguirne il test su una nuova. I risultati raccolti si riferiscono alla seconda prova dove non si è avuto nessun malfunzionamento.

Test Select

Per questa prova sono stati riportati a duemila i record nella tabella well ed è stata utilizzata la seguente funzione SQL:

```
SELECT * FROM well
LEFT JOIN users ON users.id = well.creationUser
LEFT JOIN type ON type.id = well.type
WHERE well.creationUser = {0} AND well.type < {1}
```

In questo caso la query risulta composta da due *join* in vista di quelle che il *database* dovrà eseguire nell'applicazione per la ricerca delle immagini. Per evitare i problemi di *caching* venivano modificati i valori di *creationUser* e *type*.

I risultati sono stati raccolti nel seguente grafico:

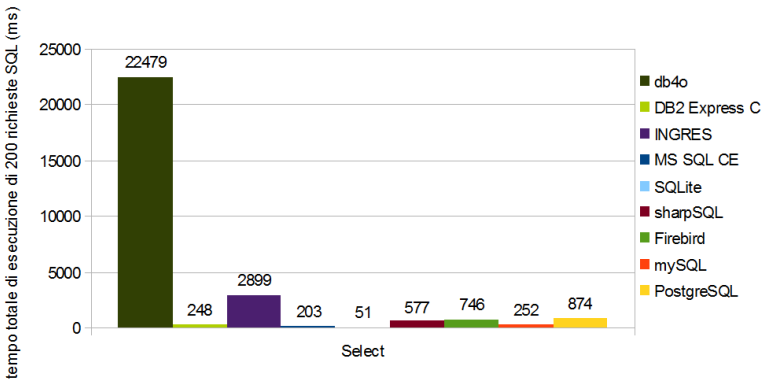


Figura 2.5: Grafico risultati del test di Select

I *database* sono statii divisi in tre gruppi e raccolti in questa tabella in base al tempo di riposta ad una singola *request*:

< 2 ms		< 5 ms		> 5 ms	
SQLite	0,3 ms	sharpSQL	2,9 ms	INGRES	14,5 ms
MS SQL CE	1 ms	Firebird	3,5 ms	DB4O	112,4 ms
DB2 Exp. C	1,2 ms	Postgres SQL	4,4 ms		
MySQL	1,3 ms				

2.3 Motivazione della scelta effettuata

Nella scelta del *database* da utilizzare per l'applicazione da sviluppare sono quindi state considerate sia le caratteristiche dei singoli che le loro prestazioni e i *database* migliori sono stati raccolti nella seguente tabella.

database	tempo select	tempo insert	gratuito
SQLite	**0,3ms**	6,6ms	**x**
Postgres SQL	4,4ms	1ms	**x**
DB2 Exp. C	1,2ms	1ms	
sharpSQL	2,9ms	**0,18ms**	**x**
MS SQL CE	1ms	0,36ms	

L'analisi della tabella indica che per un'applicazione distribuita la scelta migliore risulta essere il DB2 Express C, versione gratuita del DB2 prodotto

dalla IBM, purtroppo però per una distribuzione commerciale non è utilizzabile. Per un applicativo locale invece l'insieme di caratteristiche e prestazioni indica sharpSQL e MS SQL CE come i migliori ma il primo ha dimostrato di non essere stabile mentre il secondo ha bisogno dell'acquisto di una licenza.

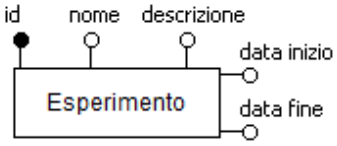
La scelta ricade quindi su Postgres SQL o su SQLite, entrambi grautiti per scopi commerciali. Il primo è un *database* molto avanzato ed ha la possibilità di essere usato anche da remoto, il secondo è molto più semplice, locale e che quindi può essere integrato all'interno dell'applicazione. Dalle prestazioni si nota che SQLite è molto più perfomante nell'esecuzione di *query* SELECT, funzione fondamentale per la nostra ricerca delle immagini. Per questi motivi il *database* scelto per l'applicazione è stato SQLite.

Capitolo 3

Analisi ed implementazione

3.1 Analisi dei requisiti

Attraverso una serie di colloqui con il gruppo di ricercatori di biologia cellulare che saranno gli utilizzatori finali di questo lavoro, sono state evidenziate tutte le specifiche utili alla definizione e modellazione della basi di dati. In particolare si è analizzato nello specifico il modo di operare dei biologi durante la definizione ed attuazione di un esperimento di biologia cellulare tenendo in particolare considerazione il flusso delle informazioni trattate e il loro utilizzo in termini di rierca. Nel *database* andranno inserite tutte le proprietà e le informazione riguardanti un'esperimento e le immagini acquisite. Per la sua creazione sono state individuate diverse entità con i relativi attributi. La prima entità rilevata è quella riguardante proprio l'esperimento.



Questa entità contiene gli attributi dell'esperimento quali il nome, la descrizione, la data di inizio e di fine. Ogni esperimento poi sarà individuabile attraverso un identificativo univoco (id).

Un'esperimento utilizza uno o più pozzetti nei quali vengono messe in coltura le cellule, la cui entità è così definita.

Questa entità contiene gli attributi del pozzetto quali l'identificativo formato da un numero e da un codice, la descrizione, l'informazione per sapere se si tratta del solo terreno della coltura (*background*), l'area che sarà espressa in cm^2, la data di inoculazione del fluorocromo (probe) per le analisi in fluorescenza e la data eventuale di contaminazione. Ogni pozzetto poi sarà individuabile attraverso un identificativo univoco (id).

Esperimento e pozzetti sono legati dalla relazione.

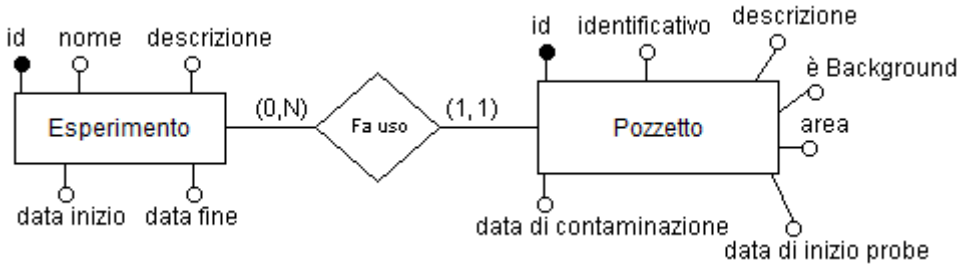
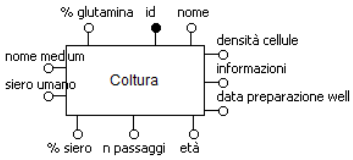


Figura 3.1: Relazione esistente tra esperimento e pozzetto.

I pozzetti utilizzano uno specifico terreno di cultura, la cui entità è così definita.



Questa entità contiene gli attributi della coltura quali il nome, la densità delle cellule, l'età della coltura,eventuali informazione testuali annotate dall'utente, il numero di passaggio effettuati, la data in cui sono stati preparati i pozzetti, la percentuale di siero e se si tratta di siero umano, il nome del medium e la percentuale di glutamina. Ogni coltura è identificata attraverso un identificativo univoco (id).

Colture e pozzetto sono legati dalla relazione.

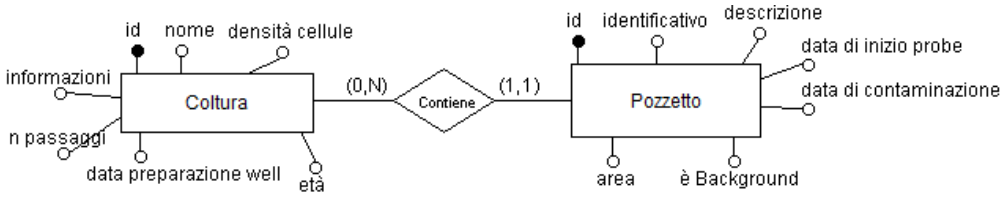
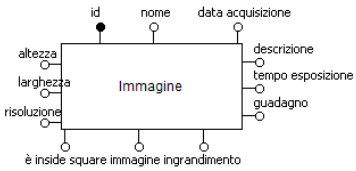


Figura 3.2: Relazione esistente tra coltura e pozzetto.

Dei vari pozzetti dell'esperimento saranno prese delle immagini, la cui entità è così definita.



Questa entità contiene gli attributi dell'immagine quali il nome, l'immagine stessa, la data di acquisizione, la descrizione delle immagine, il tempo d'esposizione, il guadagno, il fattore d'ingrandimento del miscroscopio, la risoluzione in $\mu m/pixel$, se l'immagine è stata presta all'interno di un riferimento posto sul pozzetto (square) e la dimensione dell'immagine in pixel. Ogni immagine è identificata attraverso un identificativo univoco (id).

Immagini e pozzetto sono legati dalla relazione.

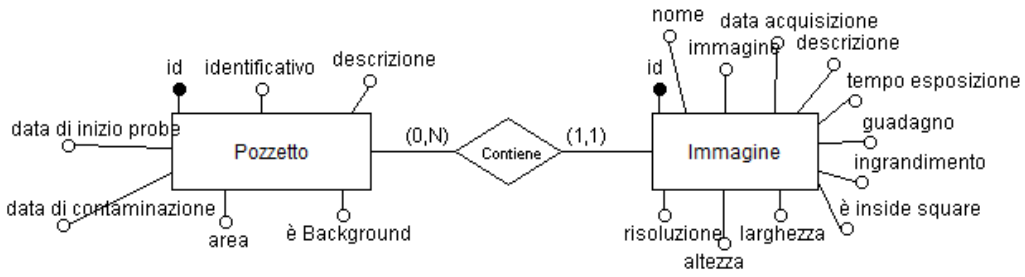
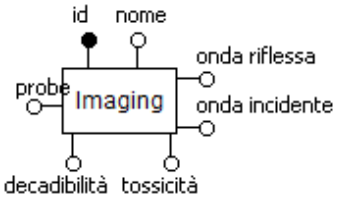


Figura 3.3: Relazione esistente tra pozzetto e immagine.

Ogni immagine viene acquisita attraverso uno strumento di imaging, la cui entità è così definita.



Questa entità contiene gli attributi dell'imaging quali il nome, la lunghezza d'onda incidente e la lunghezza d'onda riflessa espressa in nm, l'eventuale tossicità nella coltura generata dallo strumento, l'eventuale utilizzo di un particolare probe e il suo tempo di decadenza. Ogni imaging è identificata attraverso un identificativo univoco (id).

Immagine e imaging sono legate dalla relazione:

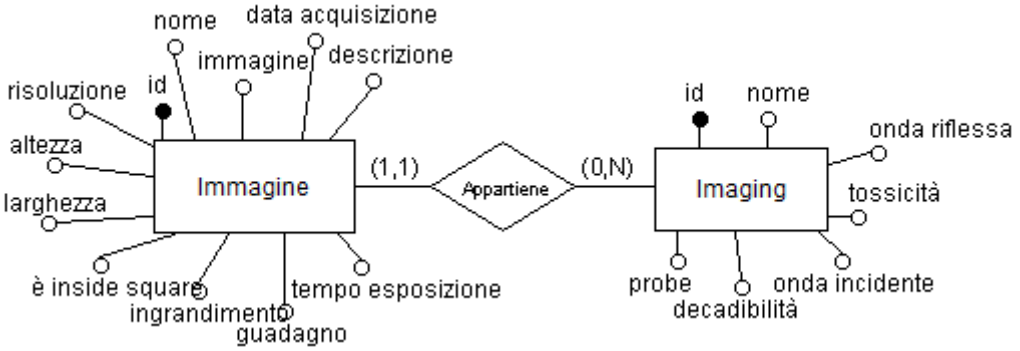
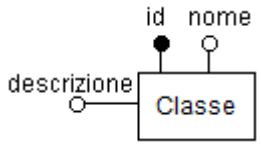


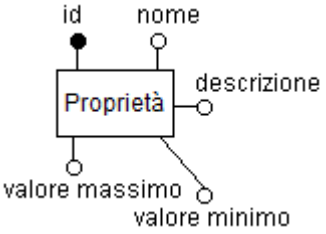
Figura 3.4: Relazione esistente tra imamgine e imaging.

Invece di configurare attributi specifici nell'immagine, viene lasciata la possibilità all'utente di generare le poprietà a seconda dell'esperimento. É stato quindi progettato una parte di metadati utlizzando le entità classe e poprietà e la relazione che lega quest'ultima con le immagini. Come vedremo questo parte di diagramma verrà tradotto nello schema relazionale in tre tabelle. Ad ogni immagine è poi possibile attribuire una proprietà, appartenente ad una classe. L'entità della classe è così definita:



Questa entità contiene gli attributi della classe quali il nome e la descrizione, ognuna è identificata attraverso un identificativo univoco (id).

Ad ogni classe possono appartenere delle proprietà, la cui entità è così definita.



Questa entità contiene gli attributi della proprietà quali il nome e la descrizione, il valore minimo e il valore massimo. Ogni proprietà è identificata attraverso un identificativo univoco (id).

La relazione che lega classe e proprietà è quindi la seguente:

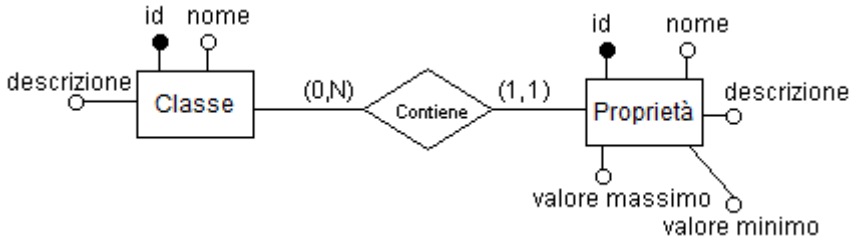


Figura 3.5: Relazione esistente tra classe e proprietà.

Tra proprietà ed immagini sussiste quindi la relazione:

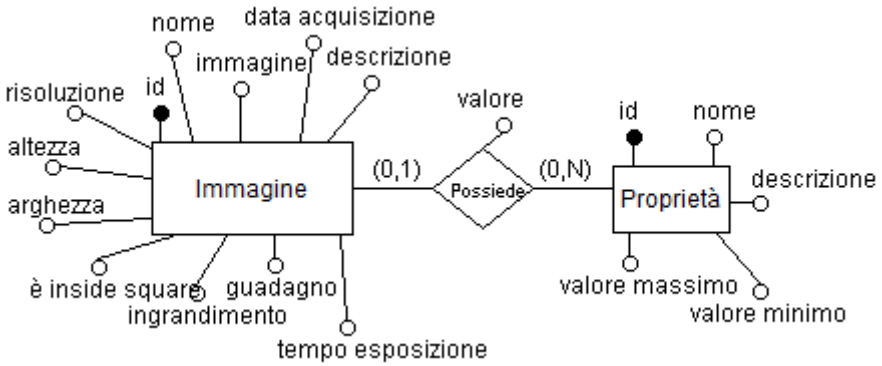


Figura 3.6: Relazione esistente tra immagine e proprietà.

Nell'assegnare una proprietà ad un'immagine si definisce l'attributo della relazione, il valore che l'immagine possiede per una generica proprietà.

Questa configurazione consente anche di poter estenedere l'applicazione e con l'utilizzo un programma automatico di elaborazione delle immagini capace di interfacciarsi col *database*, di poter introdurre dinamicamente in

queste proprietà *features* legate alle proprietà semantiche delle immagini per consentire ad esempio funzionalità di db *image retrieval*. In questo modo sarà consentito in futuro ai biologi, sviluppati gli algoritmi di estrazione delle proprietà delle immagini, di effettuare *query* per immagini.

Il Diagramma ER completo dell'applicazione è mostrato in figura 3.7.

Effettuata l'analisi per il passaggio al nuovo *database* si è pensato a come strutturare la ricerca per le immagini. Questa parte dell'applicazione doveva essere indipendete dalla struttura del db, in questo modo nel caso ci possano essere modifiche allo schema, la ricerca si può adattare e funzionare senza andare a modificare il codice. Prima della ricerca, per aiutare l'utente, è stato studiato l'inserimento di un *form* per la selezione dei campi di ricerca, in questo modo è possibile effettuare una selezione delle voci sulle quali si vuole ricercare le immagini. In questa parte è sorto il problema di come associare ogni colonna presente nel *database* con una descrizione comprensibile per l'utente e si è deciso di utilizzare un file, che dovrà venire modificato ad ogni cambiamento del db, che ripoterà per ogni colonna la singola descrizione nel formato:

nome tabella.nome colonna, descrizione, tipo

Eseguita la selezione dei campi, che l'utente potrà poi modificare ad ogni momento, si è pensato alla struttura della pagina di ricerca. Questa dovrà contere una sezione dei risultati, una funzione per l'esportazione, una di salvataggio del filtro e una per il carimento.

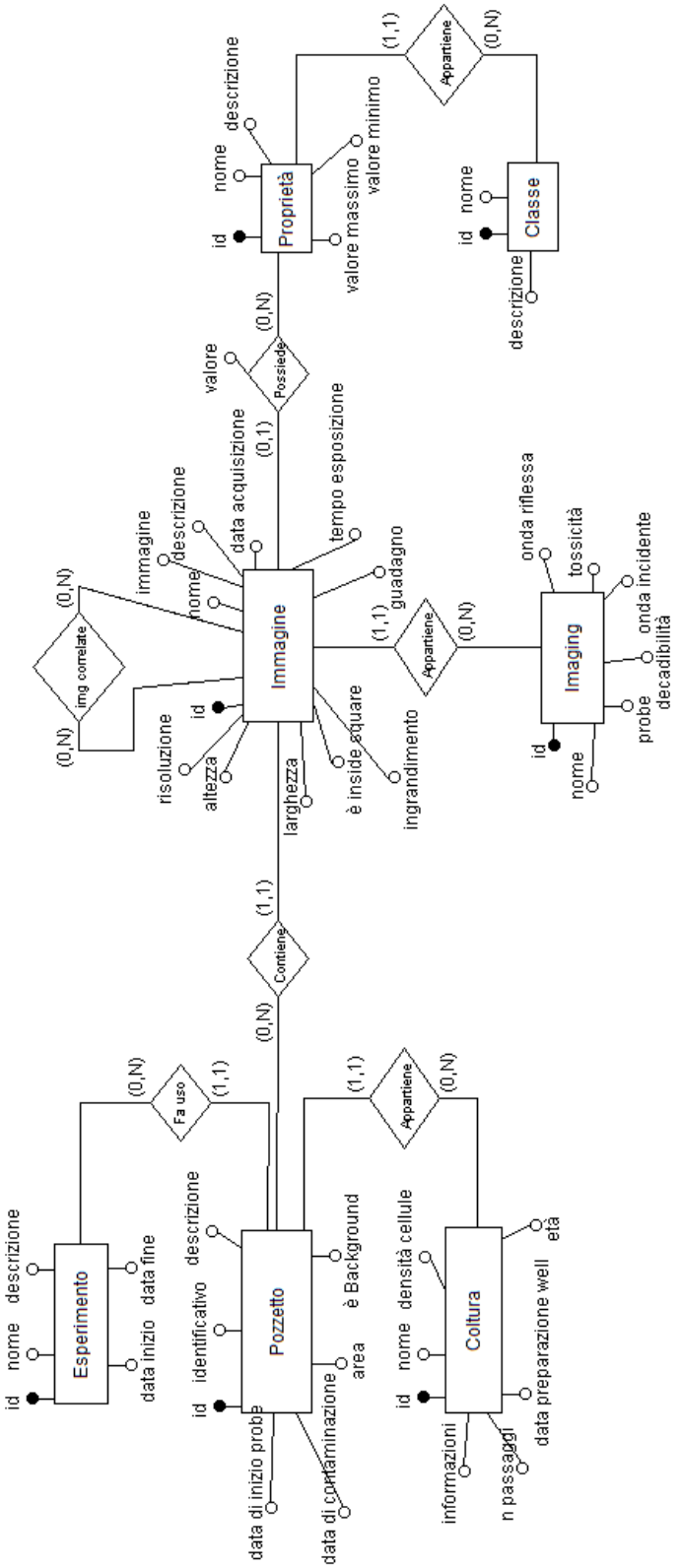


Figura 3.7: Diagramma ER del *database*

3.2 Definizione dello schema relazionale

Lo schema finale relativo all'intera applicazione è mostrato in figura 3.8.

Lo schema relazionale ottenuto dal diagramma ER è stato sviluppato in terza forma normale (3FN). Per normalizzazione di un *database* si intende il processo svolto all'eliminazione della rindondanza e del rischio di incoerenza del db [5]. Ne esistono di vari livelli che ne certificano la qualità dello schema relazionale. La terza forma normale da noi ottenuta deve rispondere oltre al suo requisito anche a quelli per la seconda (2FN) e prima forma normale (1FN).

Una base dati viene definita in 1FN se per ogni relazione contenuta questa:

1. non presenta gruppi di attributi che si ripetono cioè ciascun attributo deve essere definito su un dominio con valori atomici.

2. esiste una chiava primaria cioè esiste un'insieme di attributi che identificano in modo univco ogni tupla della relazione.

Per avere il *database* in 2FN è necessario che:

1. sia già in 1FN.

2. tutti i campi non chiave devono dipendere dall'intera chiave primaria e non solo da una parte di essa.

Infine per averlo in 3FN è necessario che:

1. sia già in 2FN.

2. tutti gli attributi non chiave dipendano direttamente dalla chiave, quindi non devono esserci attributi non chiave che dipendano da altri attributi non chiave.

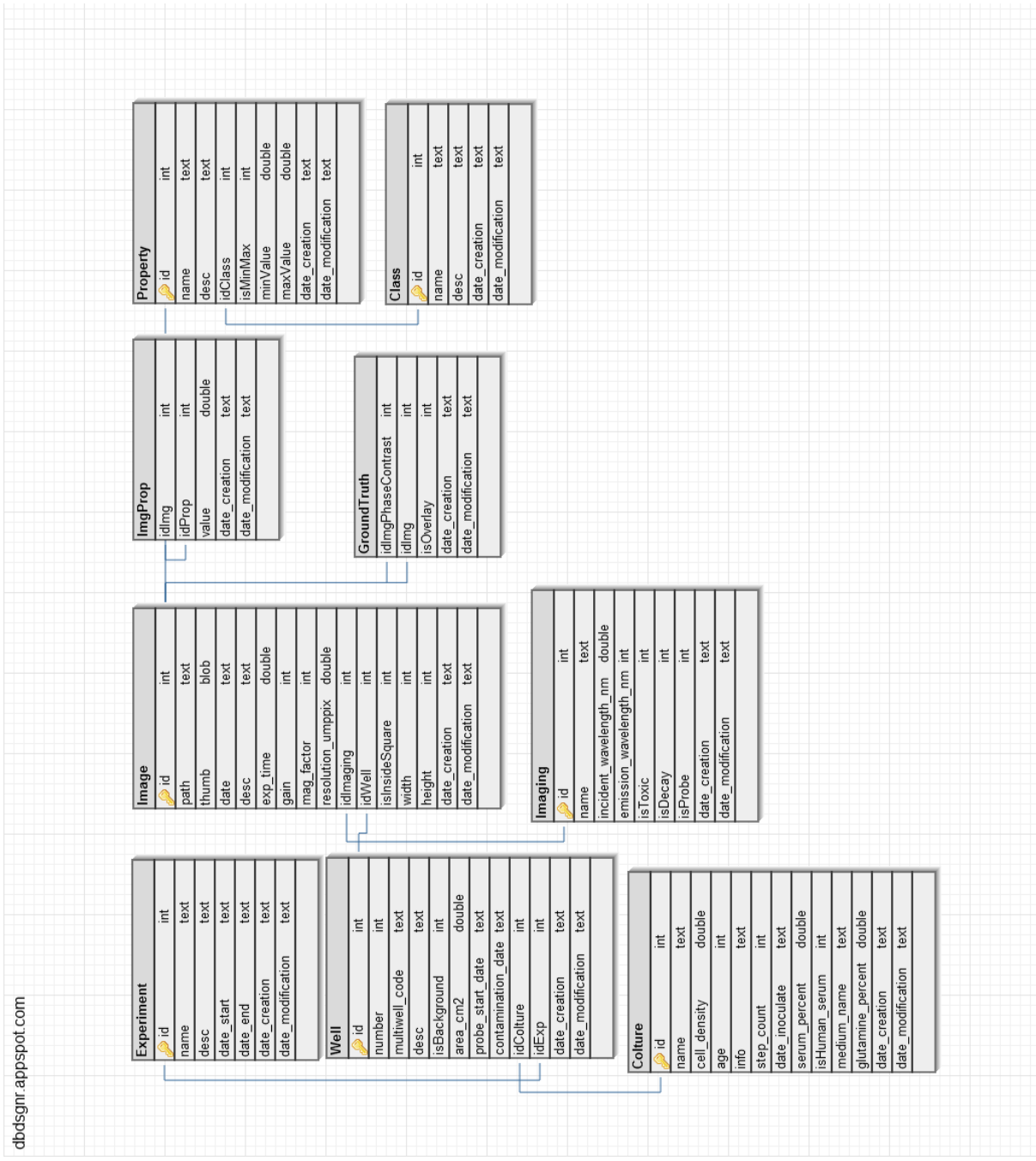


Figura 3.8: Schema relazionale del *database*

Per creare le varie tabelle sono state usate le seguenti istruzioni SQL. La tabella Experiment è così composta:

```
CREATE TABLE Experiment(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        desc TEXT,
        date_start TEXT,
        date_end TEXT,
        date_creation TEXT,
        date_modification TEXT
);
```

In ogni tabella sono stati inseriti i campi di data ultima modifica e data creazione per rendere possibile la visualizzazione degli ultimi sei esperimenti modificati o creati e per consentire in futuro di implementare nuove viste contenenti gli ultimi elementi modificati.

La tabella Colture è così composta:

```
CREATE TABLE Colture(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        cell_density REAL,
        age NUMERIC,
        info TEXT,
        step_count NUMERIC,
        date_inoculate TEXT,
        serum_percent REAL,
        isHuman_serum NUMERIC,
        medium_name TEXT,
        glutamine_percent REAL,
        date_creation TEXT,
        date_modification TEXT
);
```

La tabella Well è così composta:

```
CREATE TABLE Well(
        id INTEGER PRIMARY KEY  AUTOINCREMENT,
        number NUMERIC,
        multiwell_code TEXT,
        desc TEXT,
        isBackground NUMERIC,
        area_cm2 REAL,
        probe_start_date TEXT,
        contamination_date TEXT,
        idColture NUMERIC,
        idExp NUMERIC,
        date_creation TEXT,
        date_modification TEXT,
        foreign key (idColture) references Colture(id),
        foreign key (idExp) references Experiment(id)
);
```

Il nome del pozzetto viene diviso in due parti ed è composto dalla colonna number e multiwell_code. In base alle relazioni presenti nel diagramma ER per l'entità well, nella tabella vi sono le referenze per le tabelle Experiment e Colture.

La tabella Imaging è così composta:

```
CREATE TABLE Imaging(
        id INTEGER PRIMARY KEY  AUTOINCREMENT,
        name TEXT,
        incident_wavelength_nm REAL,
        emission_wavelength_nm REAL,
        isToxic NUMERIC,
        info TEXT,
        isDecay NUMERIC,
        isProbe NUMERIC,
        date_creation TEXT,
        date_modification TEXT
);
```

La tabella Image è così composta:

```
CREATE TABLE Image(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        path TEXT,
        name TEXT,
        thumb BLOB,
        date TEXT,
        desc TEXT,
        exp_time REAL,
        gain NUMERIC,
        mag_factor NUMERIC,
        resolution_umppix REAL,
        idImaging NUMERIC,
        idWell NUMERIC,
        isInsideSquare NUMERIC,
        width NUMERIC,
        height NUMERIC,
        date_creation TEXT,
        date_modification TEXT,
        foreign key (idImaging) references Imaging(id),
        foreign key (idWell) references Well(id)
);
```

Come si nota dalla tabella nel *database* non viene inserita l'immagine ma viene l'indirizzo in cui si trova, il nome ed una miniatura (*thumbnail*) creata dall'applicazione. In base alle relazioni presenti nel diagramma ER troviamo le referenze alle tabelle Imaging e Well.

La tabella GroundTruth è così composta:

```
CREATE TABLE GroundTruth(
        idImgPhaseContrast NUMERIC,
        idImg NUMERIC,
        isOverlay NUMERIC,
        date_creation TEXT,
        date_modification TEXT,
        foreign key (idImgPhaseContrast) references Image(id),
        foreign key (idImg) references Image(id)
);
```

Questa tabella corrisponde alla relazione "img correlate" del diagramma ER, in idImgPhaseContrast viene inserito l'identificativo (id) dell'immagine che si vuole correlare, mentre in idImg l'id dell'immagine corrente. Il campo isOverlay indica l'apertura dell'immagine in *alpha-blending*(si veda sezione 3.4) o meno.

La tabella Class è così composta:

```
CREATE TABLE Class(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        desc TEXT,
        date_creation TEXT,
        date_modification TEXT
);
```

La tabella Property è così composta:

```
CREATE TABLE Property(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        desc TEXT,
        idClass NUMERIC,
        isMinMax NUMERIC,
        minValue REAL,
        maxValue REAL,
        date_creation TEXT,
        date_modification TEXT,
        foreign key(idClass) references Class(id)
);
```

In base alla relazione tra Classe e Proprietà presente nel diagramma ER troviamo in questa tabella la referenza alla tabella Class.

La tabella ImgProp è così composta:

```
CREATE TABLE ImgProp(
        idImg NUMERIC,
        idProp NUMERIC,
        value REAL,
        date_creation TEXT,
```

```
        date_modification TEXT,
        foreign key (idImg) references Image(id),
        foreign key (idProp) references Property(id)
);
```

Questa tabella corrisponde alla relazione "Possiede" tra Proprietà e Immagine presente nel diagramma ER. L'insieme delle tabelle Classe-Proprietà-ImgProp formano la parte di *metadati* .

3.3 Trasformazione dei dati

Nell'applicazione preesistente il *database* utilizzato era MS SQL CE versione locale e compatta di Microsoft SQL Server. È un db relazionale e offre un robusto *data source*, un'ottimizzatore di *query* e una connessione affidabile. Può venire usato per sviluppare prodotti sia desktop che mobile grazie al fatto che risiede in un'unico di file di dimensione massima pari a 4 GByte e viene eseguito *in-process*. Il db scelto per sviluppare la nuova applicazione, SQLite è anch'esso un database relazionale. Incorporato un'unico file, può raggiungere dimensioni fino a 2 TByte e viene incorporato nel programma stesso. SQLite supporta quasi tutto lo standardi SQL92[1] tolte alcune caratteristiche quali: GRANT e REVOKE, ALTER TABLE (sono supportati solamente RENAME TABLE e ADD COLUMN), RIGHT OUTER JOIN, le transazioni che non possono essere nidificate e il TRIGGER è presente con alcune limitazioni.

I tipi dei dati gestiti dai *database* variano da prodotto a prodotto.Quelli definiti in MS SQL CE, corrispondenti a quelli di SQL Server con alcune eccezioni, possono venire gestiti diversamente da SQLite e si è quindi passato tabella per tabella alla loro riassegnazione [8].

SQLite definisce cinque classi di dati:

- Null: il valore è un null.
- Integer: il valore è un intero, viene memorizzato in un insieme di *byte* (da 1 a 8) a seconda della grandezza.
- Real: il valore è un numero a virgola mobile, memorizzato in 8 *byte*.

[1]Si tratta della terza revisione dello standard SQL avvenuta nel 1992, l'ultima risale al 2011 ed è la SQL-2011

- Text: il valore è una stringa, viene memorizzata utilizzando l'*encoding* del *database* (UTF-8, UTF-16BE o UTF-16LE)
- Blob: il valore è un insieme di dati e viene memorizzato così come viene dato in *input*.

Altre classe di dati, gestite da MS SQL CE con classe apposite, vengono gestite da SQLite utilizzando una di queste. I dati booleani vengono memorizzati utilizzando la classe Integer con il valore 0 per il *false* e 1 per il *true*. Le date non hanno una loro classe in SQLite ma attraverso la *Date And Time Function* presente in SQLite, è possibile memorizzarle con le classi Text, Real o Integer. La classe Text memorizza la data come un stringa secondo l'ISO8601[2] (yyyy-MM-dd hh:mm:ss.sss), con la classe Real il dato viene salvato come numeri di giorni Giuliani, cioè il numero di giorni a partire dalla mezzanotte di Greenwich del 24 novembre 4714 prima di Cristo e con la classe Integer il dato viene memorizzato come Unix Time, il numero di secondi a partire dalla mezzanotte del primo gennaio 1970.

SQLite introduce il concetto di affinità sulle colonne per massimizzare la compatibilità tra SQLite e gli altri *database*.Questo concetto consiste nel fatto che ogni colonna può memorizzare qualsiasi tipo di dato, solamente che la colonna gestirà il tipo di dato secondo l'affinità scelta. Si possono associare ad ogni colonna cinque tipi di affinità:

- Text
- Numeric
- Integer
- Real
- None

Una colonna con affinità Text memorizzerà tutti i dati utilizzando le classi Null, Text o Blob. Se un numero viene inserito in una colonna Text, questo sarà convertito in testo prima di essere memorizzato.
Una colonna con affinità Numeric può contenere valori utilizzando tutti i tipi i classe. Quando un testo viene inserito in una colonna Numeric questo

[2]L'ISO 8601 conosciuto anche come *Data elements and interchange formats - Information interchange - Reppresentation of dates and times* è uno standard internazionale per la rappresentazione di date e orari. Fu pubblicato la prima volta nel 1988, l'ultima revisione risale al 2004.

viene convertito in Integer o Real a seconda della preferenza. La conversione tra Text e Real è senza perdite e reversibile fino alla quindicesima cifra significativa, se la conversione da Text a Real o Integer non è possibile, il dato verrà memorizzato utilizzando la classe Text.
L'affinità Integer funziona alla stessa maniera di quella numeric con la differenza che il dato sarà memorizzato utilizzando la calsse Integer, stessa cosa con l' affinità Real. Una colonna con affinità None non ha alcuna preferenza sulla classe da utilizzare per la memorizzazione del dato.

L'affinità di una colonna viene determinata dal tipo di dato che si vuole salvare sul *database* secondo le seguenti regole:

1. Se la colonna viene definita col tipo di dato INT allora sarà assegnata l'affinità Integer.

2. Se la colonna viene definita con un tipo di dato che contiene le parole CHAR, CLOB o TEXT la colonna avrà affinità Text.

3. Se la colonna viene definita con BLOB o non viene specificato nulla, allora avrà affinità None.

4. Se la colonna viene definita con un tipo di dato contenente FLOA, DOUB o REAL la colonna avrà affinità Real.

5. In tutti gli ltri casi la colonna avrà affinità Numeric.

Un problema emerso durante la conversione del *database* da MS SQL CE a SQLite è stata la gestione delle colonne di tipo DATETIME. Con le regole precedenti queste vengono gestite con affinità Numeric quindi salvate o come numero di giorni Giuliani o come secondi. In questo caso si è deciso di definire tutte le colonne contenenti una data di tipo Text, in questo modo si è reso il *cast* da *database* all'applicazione molto più semplice.

3.4 Nuove funzionalità

La parte esistente di applicazione è stata reingegnerizzata, questo lavoro oltre al cambio di *database*, è consistito nella riscrittura di parte del codice. Sono inoltre state implementate nuove funzioni e riprogettata la parte grafica per renderla *user friendly*.

Durante l'adattamento dello schema alla nuova base dati si è provveduto ad inserire in ciascuna tabella due colonne: data di creazione e data di

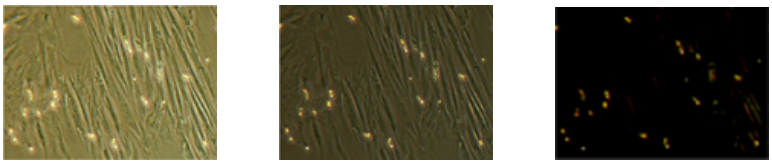
ultima modifica. Con l'aggiunta di questi nuovi dati sarà possibile in futuro, se richiesto, creare elenchi delle ultime operazioni effettuate o come vedremo in seguito, sono state utili per la riprogettazione della parte grafica.

Un'altra funzionalità introdotta nel nuovo sistema riguarda la presentazione della scheda immagini. In questa scheda si ha la possibilità di relazione diverse immagini appartenenti al pozzetto. Questa funzionalità risulta particolarmente utile quando vengono acquisite immagini con diversi strumenti di imaging della stessa zona del pozzetto. Per agevolare la comprazione delle zone di interesse, si è sviluppato uno strumento di visualizzazione che fa uso dell' *alpha blending*.

L'*alpha blending* consiste nel miscelare insieme due immagini in *overlay* secondo un preciso algoritmo. Si tratta di una combinazione convessa cioè una combinazione lineare di elementi fatta con coefficenti non negativi. Il coefficente *alpha* da cui dipende il risultato dell'operazione varia da un valore 0.0 a 1.0 dove 0.0 rappresenta la totale trasparenza del pixel e 1.0 la totale opacità. Avremo che:

$$\vee p_1, p_2 \in I_1, I_2 \qquad p = \alpha p_1 + (1 - \alpha) p_2$$

Dove p_1 e p_2 sono i pixel delle rispettive immagini I_1, I_2.



(a) *alpha* = 1 (b) *alpha* = 0.5 (c) *alpha* = 0

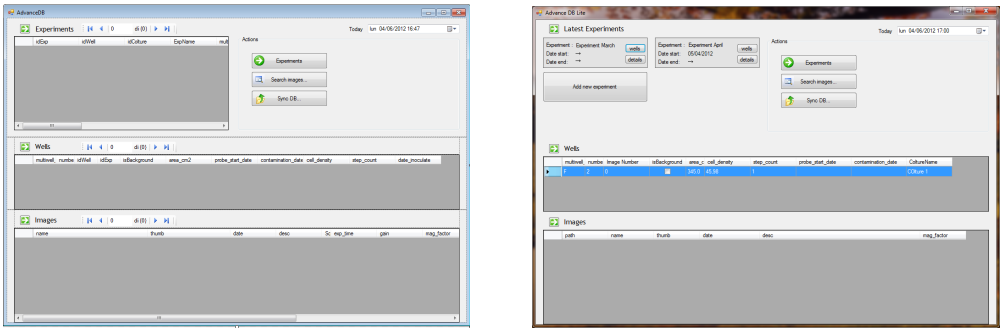
Figura 3.9: Esempio di *alpha blending*

Altra funzione aggiunta è la possbilità di definire un valore limite massimo e minimo nelle proprietà da attribuire alle immagini. Questi limiti, oltre ad offrite maggiori informazioni, saranno utili all'utente diminuendo la probabilità d'inserimento di un valore errato.

Un'altro aspetto analizzato in questa fase è stato il mancato ritrovamento del file contenente il *database* o il suo malfunzionamento all'avvio dell'applicazione. Sono state trovate due soluzioni a questo problema: la cancellazione dell'istanza corrotta, se presente, e la creazione di una nuova

oppure l'uso di un db esistente, ad esempio una copia di *backup*. Entrambe le situazioni sono state implementate e la scelta è stata lasciata all'utente.

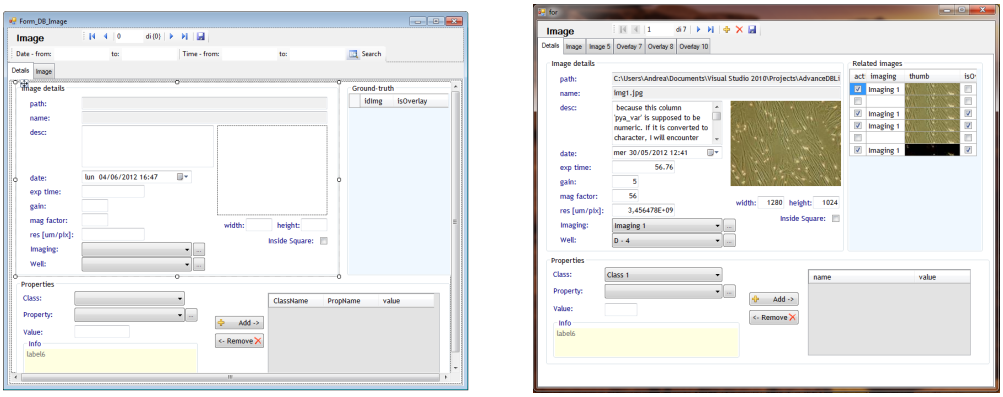
La riprogettazione della parte grafica è partita dalla pagina iniziale, dove si è passati da una versione tutta tabellata ad una visione degli ultimi sei esperimenti inseriti o modificati con la possibilità di vedere in anteprima i pozzetti e le immagini in essi contenuti nelle tabelle sottostanti premendo il tasto *well*, oppure aprire una pagina contenente i vari dettagli dettagli premendo il tasto *details*.



(a) Vecchia grafica (b) Nuova grafica

Figura 3.10: Vecchia e nuova grafica della *landing* form

Oltre a queste vista, anche la *form* relativa alle immagini è stata rivista nella parte riguardante l'*overlay* delle immagini (Figura 3.11). Inizialmente infatti la gestione era molto complessa e non facilmente intuibile dall'utente, consisteva nel trascinare una *thumbnail* di un'immagini nella relativa tabella all'interno della finestra in cui la si voleva visualizzare. Questa procedura è stata sostituita con la visualizzazione nella tabella, precedentemente predisposta per il *drag and drop* delle immagini, delle miniature di tutte le immagini appartenente allo stesso pozzetto e all'informazione dell'imaging di appartenza, oltre all'inserimento di due comandi per l'aggiunta dell'immagine in sola visualizzazione o in *overlay* con *alpha blending*.



(a) Vecchia grafica (b) Nuova grafica

Figura 3.11: Vecchia e nuova grafica della *Image form*

3.5 Implementazione

In questa parte viene mostrato il metodo che si è seguito per implementare l'applicazione, le parti più importanti e i problemi emersi. Sono stati implementati tutti gli oggetti delle relative tabelle in modo che la loro gestione, all'interno dell'applicazione, risulti più semplice. Ad esempio per la tabella well è stata creata la classe Well.cs. All'interno di questa classe viene definito l'oggetto *Well*:

```
private int id;
private int number;
private string desc;
private char multiwellCode;
private Boolean isBackground;
private Double areaCm2;
private DateTime probeStartDate;
private DateTime contaminationDate;
private int idColture;
private int idExp;
private DateTime creationDate;
private DateTime modificationDate;

public Well() { }

public Well(int id, int number, char multiwellCode, string desc, Boolean isBackground,
    Double areaCm2, DateTime probeStartDate, DateTime contaminationDate, int idColture,
    int idExp, DateTime creationDate, DateTime modificationDate)
{
    this.id = id;
    this.number = number;
    this.desc = desc;
    this.multiwellCode = multiwellCode;
    this.isBackground = isBackground;
    this.areaCm2 = areaCm2;
    this.probeStartDate = probeStartDate;
    this.contaminationDate = contaminationDate;
    this.idColture = idColture;
    this.idExp = idExp;
    this.creationDate = creationDate;
    this.modificationDate = modificationDate;
}
```

Figura 3.12: Campi e definizione del costruttore Well

Ogni campo avrà poi il relativo *getter* e *setter*. Per ogni classe di questo tipo è stata implementata anche una classe che ne gestisce l'iterazione col *database*.

Per l'oggetto Well è stata definita la classe WellMethods.cs. All'interno troviamo il metodo per l'inserimeto di una well:

```
public static Well insertWell(SQLiteConnection sqlConn, Well well)
{
    if (sqlConn.State != ConnectionState.Open) sqlConn.Open();
    SQLiteCommand cmd = sqlConn.CreateCommand();
    int isBackground = 0;
    if (well.IsBackground == true) isBackground = 1;
    cmd.CommandText = string.Format(Well.INSERT_ELEMENT, well.Number, well.MultiwellCode,
        well.Desc, isBackground, well.AreaCm2, well.ProbeStartDate.ToString("dd-MM-yyyy HH:mm:ss"),
        well.ContaminationDate.ToString("dd-MM-yyyy HH:mm:ss"), well.IdColture, well.IdExp,
        DateTime.Now.ToString("dd-MM-yyyy HH:mm:ss"), DateTime.Now.ToString("dd-MM-yyyy HH:mm:ss"));
    cmd.ExecuteNonQuery();
    cmd.CommandText = Well.GET_LAST_ID;
    well.Id = Int32.Parse("" + cmd.ExecuteScalar());
    return well;
}
```

Figura 3.13: Metodo per l'inserimento di un record well nel *database*

Questo metodo ritorna l'oggetto well inserito valorizzandone anche il campo identificatore (id). Nel caso la connessione risultasse chiusa, viene aperta e la stringa contenente l'istruzione SQL viene riempita cn le varie proprietà del pozzetto inserite dall'utente, in particolare le date vengono trasformate in stringa con un pattern definito.

Altro metodo presente è quello per l'aggiornamento di un record presente:

```
public static void updateWell(SQLiteConnection sqlConn, Well well)
{
    if (sqlConn.State != ConnectionState.Open) sqlConn.Open();
    SQLiteCommand cmd = sqlConn.CreateCommand();
    int isBackground = 0;
    if (well.IsBackground == true) isBackground = 1;
    cmd.CommandText = string.Format(Well.UPDATE_BY_ID, well.Number, well.MultiwellCode,
        well.Desc, isBackground, well.AreaCm2, well.ProbeStartDate.ToString("dd-MM-yyyy HH:mm:ss"),
        well.ContaminationDate.ToString("dd-MM-yyyy HH:mm:ss"), well.IdColture, well.IdExp,
        DateTime.Now.ToString("dd-MM-yyyy HH:mm:ss"), well.Id);
    cmd.ExecuteNonQuery();
}
```

Figura 3.14: Metodo per l'aggiornamento di un record well nel *database*

Anche in questo metodo la connessione viene aperta se risulta chiusa e vengono passati i campi aggiornati del pozzetto.

Infine un'altro metodo sempre presente in questo tipo di classi è quello per la cancellazione di un record:

```
public static void deleteWellById(SQLiteConnection sqlConn, int idWell)
{
    if (sqlConn.State != ConnectionState.Open) sqlConn.Open();
    SQLiteCommand cmd = sqlConn.CreateCommand();
    cmd.CommandText = string.Format(Well.DELETE_BY_ID, idWell);
    cmd.ExecuteNonQuery();
}
```

Figura 3.15: Metodo per la cancellazione di un record well nel *database*

In questo caso per cancellare un record well inserito nel *database* è sufficente passare alla funzione l'identificativo.

Per testare le classi create si è passati alla generazione della connessione col *database*. Tutta l'applicazione utilizza una sola connessione che viene passata dalla sua creazione all'avvio del programma in tutte le altre *form*. SQLite utilizza un metodo di *reader/writer locks* [12], significa che quando qualsiasi processo sta leggendo una parte del *database*, a tutti gli altri processi è impedito la scrittura su qualsiasi altra parte, se qualche processo sta scrivendo, a tutti gli altri processi è impedito di leggere. Per questo motivo viene utilizzata un'unica connessione per tutta l'applicazione. Se eventualmente la connessione dovesse chiudersi, questa verrà ristabilita. La connessione iniziale viene effettuata al caricamento della *landing form*:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    ConnectionStringSettings connString = System.Configuration.ConfigurationManager.ConnectionStrings[CONN_STRING];
    this.conn = new SQLiteConnection(connString.ConnectionString);
```

Figura 3.16: istruzioni per la creazione della connessione al *database*

La stringa di connessione per un *database* SQLite è del tipo:

```
Data Source=|DataDirectory|\ nomeDB.sqlite
```

Nel caso la connessione non riesca per mancanza del *database* o perchè corrotto viene aperta la *form* con le due opzioni: l'utilizzo di un db esistente o la creazione di uno nuovo. Nel caso venga scelto d'usare un *database* già esistente, viene copiato il file indicato nella cartella dell'applicazione e si ricrea la connessione. Nel caso in cui venga scelto di crearne uno nuovo, si crea il file vuoto e la connessione, si eseguono in serie le *query* per la creazione delle varie tabelle e si visualizza la *landing form*.

Effettuati questi passaggi si è passati alla riscrittura del codice esistente applicando le modifiche riguardanti la visualizzazione degli ultimi sei esperimenti modificati o inseriti. Dovendo contare nelle modifiche di un'esperimento anche l'inserimento o la modifica di un pozzetto a lui appartenente, si è avuto il problema di scrivere una *query* che automanticamente, senza dover effettuare ciclci su tutti gli esperimenti e pozzetti, desse come risultato i record degli esperimenti che cercavamo. La *query* utilizzata è la seguente:

```
SELECT * FROM Experiment
LEFT JOIN Well ON Well.idExp = Experiment.id
GROUP BY Experiment.id
ORDER BY (
CASE WHEN (
SELECT Experiment.date\_{}modification FROM Experiment
ORDER BY Experiment.date\_{}modification DESC LIMIT 1) >
( SELECT Well.date\_{}modification FROM Well
ORDER BY Well.date\_{}modification DESC LIMIT 1)
THEN Experiment.date\_{}modification
ELSE Well.date\_{}modification
END ) DESC LIMIT 6;
```

Dopo aver riscritto il codice delle varie finestre per adattarle al nuovo *database*, sono state implementate le nuove funzioni, se quella riguardante il limite massimo e minimo nelle proprietà è risultata semplice, meno è stato per l'*alpha blending*.

Per implementare questa funzione sono state prese le due immagini, trasformate in array di byte e si è creato l'array che conterrà i byte dell'immagine finale. Si è poi effettuato un ciclo dal primo all'ultimo byte dell'array della prima immagine e si è applicata la formula per la miscelazione. Il tutto è stato poi riconvertito in immagine e visualizzata in una nuova scheda.

```
private System.Drawing.Bitmap mergeImage(double alpha, Image gtImage)
{
    System.Drawing.Image firstImage = System.Drawing.Image.FromFile(this.image.Path + "//" + image.Name);
    System.Drawing.Image secondImge = System.Drawing.Image.FromFile(gtImage.Path + "//" + gtImage.Name);
    byte[] firstByte = getBitmapData(new Bitmap(firstImage));
    byte[] secondByte = getBitmapData(new Bitmap(secondImge));
    byte[] byteImageResult = new byte[firstByte.Length];
    int limit = 0;
    if (firstByte.Count() == secondByte.Count() || firstByte.Count() < secondByte.Count()) limit = firstByte.Count();
    else limit = secondByte.Count();
    for (int index = 0; index < limit; index++)
    {
        double color1 = firstByte[index];
        double color2 = secondByte[index];
        double res = alpha * (color1) + (1 - alpha) * (color2);
        byte result = (byte)res;
        byteImageResult[index] = result;
    }
    System.Drawing.Bitmap imageResult = new Bitmap(firstImage);
    setBitmapData(imageResult, byteImageResult);
    return imageResult;
}
```

Figura 3.17: Implementazione dell'algoritomo di *alpha blending*

Si è passati poi alla revisione della gestione delle immagini collegate. Nel sistema precedente era usato un metodo *drag and drop* dove si prendeva una *thumbnail* dall'elenco delle immagini visualizzate nella finestra dettagli degli esperimenti e la si trascinava sulla tabella "*Related Images*" dell'immagine che si voleva usare. Questo modo era poco intuitivo e *user friendly* e si è pensato di sostituirlo con la visione nella tabella "*Related Images*" di tutte le immagini appartenenti allo stesso pozzetto e, spuntando la *checkbox* relativa ad active, viene aperta una scheda con l'immagine, spuntanto quella relativa ad *Overlay*, la scheda si apre in modalità *alpha blending*.

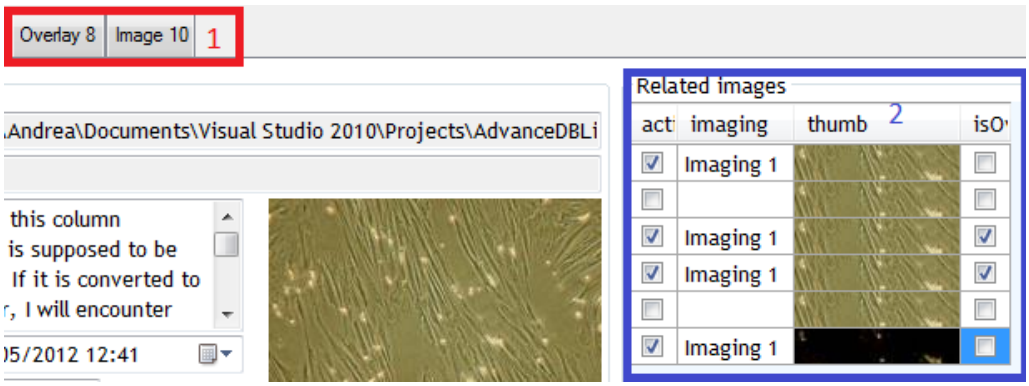


Figura 3.18: La nuova gestione delle *Related images*. (1) L'apertura delle schede con visione normale delle immaggini o in *overlay*. (2) La tabella con tutte le immagini appartenenti al pozzetto e le *checkbox* per aprire le schede.

Terminato l'adattamento della parte del sistema esistente si è eseguito il *debug* dell'applicazione.

Capitolo 4

Sviluppo dell'interfaccia di ricerca

4.1 Analisi e criteri di usabilità

La parte di ricerca delle immagini è stata implementata completamente, l'idea era quella di realizzare un'interfaccia di ricerca che fosse totalmente indipendente dalla struttura del *database* e quindi poter essere utilizzata anche per altre applicazioni senza dover apporre grandi modifiche al codice. Altra funzione che si voleva implementare era la scelta degli elementi su cui costruire il filtro, considerando d'usare un *database* molto ampio, una preselezione dei campi poteva risultare utile. Effettuata la ricerca bisognava dare la possibilità di salvarla, caricarla e di esportarla.

Requisito fondamentale della finestra di ricerca doveva essere la facilità di utilizzo, la possibilità di potersi creare un filtro utilizzando il nome delle varie proprietà specificandone la condizione (maggiore, minore, uguale, ...) ed il valore. Anche il raggruppamento delle varie condizioni doveva essere immediato e facilmente intuibile.

4.2 Definizione del layout

Si è cercato di usare sia per la finestra di ricerca che di selezione degli elementi un layout molto seplice ei facilmente intuibile. La finestra di selezione degli elementi di ricerca è formata da una parte ad albero,a sinistra, dove i vari campi sono contenuti all'interno del relativo oggetto a cui fanno riferimento,

e a destra vi sono i vari pulsanti con diverse funzionalità

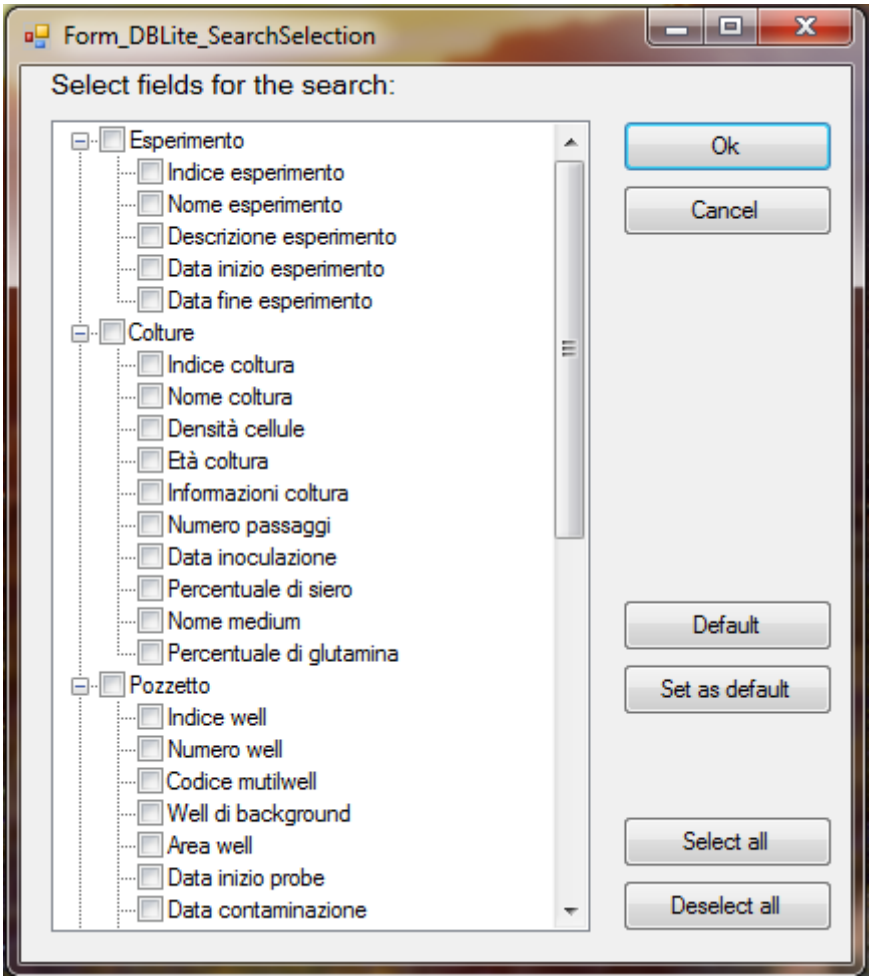


Figura 4.1: Finestra di selezione degli elementi di ricerca

Volendo rendere la finestra di ricerca molto semplice e di facile utilizzo, si è deciso di utilizzare un layout formato da una parte superiore riguardante il filtro, caricamento e salvataggio di ricerche salvate e la parte inferiore è stata destinata alla visualizzazione dei risultati. La creazione di una condizione per il filtro avviene in 3 passi:

1. Si sceglie dalla prima *combobox* il campo su cui si vuole effettuare la ricerca

2. Si sceglie dalla secondo *combobox* la condizione (maggiore, minore, ...) che si vuole utilizzare. Il contenuto delle condizioni varierà in base al tipo di campo selezionato.

3. In base al campo selezionato inseriremo il valore di ricerca, se il campo è di tipo numerico o di testo si avrà una *textbox*, se il campo è un boolean si avrà una *combobox* contenente i due valori *true* e *false*, se invece il campo è di tipo data avremo un *datetimepicker*.

Effettuati questi 3 passi è già possibile effettuare la ricerca oppure è possibile aggiungere ulteriori condizioni. Nella definizione di altre condizioni, si dovrà settare la *combobox* di connessione scegliendo se porre quella successiva in *and* o in *or*. Nel caso venga selezionato *or*, la condizione successiva sarà indentata rispetto alla precedente, in modo da rendere intuibile graficamente il raggruppamento delle condizioni.

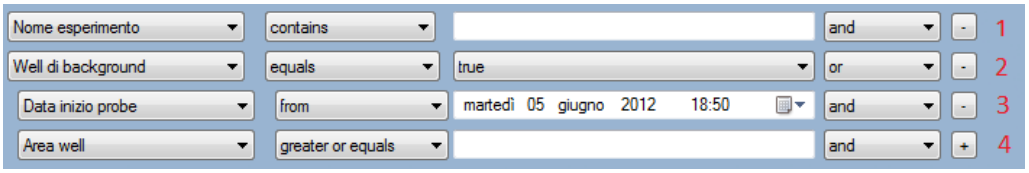


Figura 4.2: Parte della finestra di ricerca riguardante la costruzione del filtro. (1) Elemento di tipo testo, il campo dove inserire il valore è una *textbox*. (2) Elemento di tipo booleano, il campo dove inserire il valore è una *combobox*, in questo caso viene selezionato anche la condizione di mettere le prossime in *or*. (3) Elemento di tipo data, il campo di selezione valore è un *datetimepicker*. (4) Elemento di tipo numerico, il campo dove inserire il valore è una *textbox*.

In questo caso la *query* di ricerca creata risulterà:

```
SELECT * FROM image
LEFT JOIN Well ON image.idWell = Well.id
LEFT JOIN Experiment ON well.idExp = Experiment.id
WHERE  (well.isbackground = '1')
OR (well.contamination_date >= '05-06-2012 18:50:00')
GROUP BY image.id
```

Come si può notare se il campo di ricerca non viene valorizzato (esperimento o area well) la corrispondente clausola *where* non viene inclusa nella query generata. Limite di questa ricerca è la possibilità di poter effettuare *query* di un solo livello.

Sul lato sinistro del filtro è stata posta una tabella dove sono presenti le ricerche salvate, cliccando su una riga viene eseguita la ricerca e nel filtro

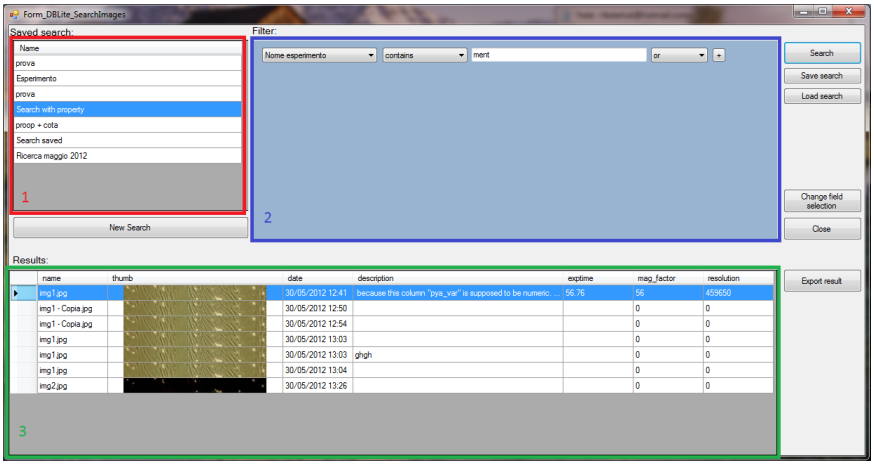


Figura 4.3: Layout della pagina di ricerca. (1) Tabella delle ricerce salvate. (2) Filtro della ricerca. (3) Risultati della ricerca.

vengono caricate le sue condizioni.

4.3 Implementazione

La scrittura del codice è iniziata dalla finestra di selezione dei campi, questa viene visualizzata prima della pagina di ricerca solamente al primo accesso, in seguito si potrà modificare la selezione attraverso il tasto apposito nel *form* di ricerca. Perchè non vi vossero i nomi delle colonne del *database* ma dei nomi comprensibili per l'utente è stato creato un file, in formato *comma-separated values*(csv), contenente, per ogni colonna, la relativa descrizione:

nome tabella.nome colonna, descrizione, tipo

In caso di modifica al *database* come l'aggiunta di una colonna, non bisognerà fare altro che aggiungere la relativa riga al file e il campo comparirà nella selezione, per eliminare una voce invece basterà togliere la riga corrispondente. Il file csv viene letto solo una volta, alla prima apertura della ricerca, dopodichè i campi vengono memorizzati all'interno di un *array* di oggetti FieldRappresentation:

Questo vettore, definito in Program.cs, la classe *Main* dell'applicazione, è quindi visibile da tutti i *form* e sarà utilizzato per costruire le *query*.

Fields Rappresentation.csv

```
experiment,Esperimento,table
experiment.id,Indice esperimento,numeric
experiment.name,Nome esperimento,text
experiment.desc,Descrizione esperimento,text
experiment.date_start,Data inizio esperimento,data
experiment.date_end,Data fine esperimento,data
colture,Colture,table
colture.id,Indice coltura,numeric
colture.name,Nome coltura,text
colture.cell_density,Densità cellule,numeric
colture.age,Età coltura,numeric
colture.info,Informazioni coltura,text
colture.step_count,Numero passaggi,numeric
colture.date_inoculate,Data inoculazione,data
colture.serum_percent,Percentuale di siero,numeric
colture.medium_name,Nome medium,text
colture.glutamine_percent,Percentuale di glutamina,numeric
well,Pozzetto,table
well.id,Indice well,numeric
well.number,Numero well,numeric
well.multiwell_code,Codice mutilwell,text
well.isbackground,Well di background,bool
```

Figura 4.4: Pezzo del file csv utilizzato per la descrizione dei campi

```
{
    [DelimitedRecord(",")]
    public class FieldRappresentation
    {
        public FieldRappresentation();
        public FieldRappresentation(string key, string value, string type);

        public string Key { get; set; }
        public string Type { get; set; }
        public string Value { get; set; }
    }
}
```

Figura 4.5: Definizione dell'oggetto contenente le informazioni dei campi

Un problema emerso durante la costruzione dell'albero contenente i campi ha riguardato l'inserimento delle proprietà che possono venire associate ad un'immagine. Queste non sono statiche, ma vengono definite dall'utente. Per risolvere questo problema, ogni volta che si apre la finestra di selezione, essendo le proprietà appartenenti a classi,se ne controlla l'esistenza ottenendone gli id, viene eseguito poi un cliclo dove, per ogni classe, si ricava la lista delle proprietà. Una volta trovate, viene creato l'oggetto FieldDescription da inserire nell'array insieme agli altri campi:

Nell'albero di selezione dei campi comparirà la proprietà nel formato :

nomeClasse.nomeProprietà

in questo modo l'utente potrà effettuare ricerche anche in base ad esse.

La creazione della finestra di ricerca è partita analizzando alcune *query* di complessità crescente:

```
List<FieldRappresentation> propertiesClassName = new List<FieldRappresentation>();
for (int i = 0; i < classId.Count; i++)
{
    Class c = ClassMethods.getClassById(sqlConn, classId[i]);
    List<int> propertiesId = PropertyMethods.getIdPropertyByIdClass(sqlConn, c.Id);
    for (int index = 0; index < propertiesId.Count; index++)
    {
        Property p = PropertyMethods.getPropertyById(sqlConn, propertiesId[index]);
        FieldRappresentation fieldrapp = new FieldRappresentation();
        fieldrapp.Key = "property." + c.Id + "." + p.Id;
        fieldrapp.Value = c.Name + "." + p.Name;
        fieldrapp.Type = Properties.Settings.Default.field_type_numeric;
        propertiesClassName.Add(fieldrapp);
    }
}
```

Figura 4.6: Ricerca delle proprietà esistenti ed inserimento nell'alberto

1. Seleziona le immagini da una certa data in poi:

   ```
   SELECT image.id FROM image
   WHERE date >= '02/03/2012';
   ```

2. Seleziona le immagini con fattore d'ingrandimento 10 e contaminate prima di una certa data:

   ```
   SELECT image.id FROM image
   LEFT JOIN well ON image.idwell = well.id
   WHERE image.mag_factor = 10
   AND well.contamination_date < '02/03/2012';
   ```

3. Seleziona le immagini con fattore d'ingrandimento 10, contaminate prima di una certa data o con una lunghezza d'onda incidente di 99nm:

   ```
   SELECT image.id FROM image
   LEFT JOIN well ON image.idwell = well.id
   LEFT JOIN imaging ON image.idImaging = imaging.id
   WHERE image.mag_factor = 10
   AND well.contamination_date < '02/03/2012'
   OR imaging.incident_wavelength_nm = 99;
   ```

Questa analisi ha fatto comprendere quali elementi erano necessari per creare dinamicamente le *query*. In primo luogo si è notato che era indispensabile conoscere le *foreign key* di ogni tabella. Si è proceduto quindi alla creazione di un'oggetto apposito per la loro gestione.

```
public class ForeignKey
{
    public ForeignKey();

    public string ForeignColumn { get; set; }
    public string ForeignTableName { get; set; }
    public string ReferredColumn { get; set; }
    public string ReferredTableName { get; set; }
}
```

Figura 4.7: Definizione dell'oggetto contenente le informazioni sulle *foreign key*

Al primo avvio della ricerca, viene riempito un'*array* contenente tutte le *foreing key* delle varie tabelle, indispensabili per l'inserimento delle *join* nelle *query*. Per popolarlo viene eseguito un ciclo sull'*array* contenente i FieldRappresentation per ricercare i campi definiti di tipo *table*:

```
List<DBComponent.ForeignKey> listKeys = new List<ForeignKey>();
foreach (FieldRappresentation field in Program.fields)
{
    if(field.Type.Equals(Properties.Settings.Default.field_type_table))
    {
        listKeys.AddRange(SearchMethods.getForeignKey(sqlConn, field.Key));
    }
}
//Inserisco le foreignkey nella variabile globale
Program.foreignKey = listKeys;
```

Figura 4.8: Ricerco i campi di tipo *table* per popolare l'array delle *foreign key*

Ottenuti i campi si ricavano le chiavi esterne utilizzando una specifica istruzione SQL:

```
//Ottengo il nome delle varie foreign key presenti in una tabella
private static string SQL_FOREIGN_KEY = " PRAGMA foreign_key_list('{0}') ";

public static List<DBComponent.ForeignKey> getForeignKey(SQLiteConnection sqlConn, string tableName)
{
    if (sqlConn.State != ConnectionState.Open) sqlConn.Open();
    SQLiteCommand cmd = sqlConn.CreateCommand();
    cmd.CommandText = string.Format(SQL_FOREIGN_KEY, tableName);
    List<DBComponent.ForeignKey> res = new List<DBComponent.ForeignKey>();
    SQLiteDataReader reader = cmd.ExecuteReader();

    while (reader.Read())
    {
        DBComponent.ForeignKey fKey = new DBComponent.ForeignKey();
        fKey.ReferredTableName = tableName;
        fKey.ReferredColumn = reader.GetString(reader.GetOrdinal("from"));
        fKey.ForeignTableName = reader.GetString(reader.GetOrdinal("table"));
        fKey.ForeignColumn = reader.GetString(reader.GetOrdinal("to"));
        res.Add(fKey);
    }
    return res;
}
```

Figura 4.9: Metodo che riporta tutte le *foreign key* di una tabella

Per la costruzione delle *query* viene utilizzato l'oggeto SelectQueryBuilder presente nel CodeEngineFramework distribuito gratuitamente. Questo oggetto permette la costruzione di stringhe di ricerca con l'inserimento delle varie *join* e *where*. Ad esempio per ottenere la prima *query* proposta nell'esempiosi dovrà iniziare definendo un'istanza del costruttore:

```
SelectQueryBuilder sqb = new SelectQueryBuilder();
```

nel quale vengono definite la tabella su cui effettuare la ricerca, in questo caso la tabella Image, e il campo che si vuole ottere:

```
sqb.SelectFromTable("image");
sqb.SelectColumn("image.id");
```

e infine si aggiungono le condizioni:

```
sqb.AddWhere("image.date", Comparison.LessOrEquals, "02/03/2012");
```

Definiti i vari campi nell'oggetto,per ottenere la *query* si utilizza il metodo *BuildQuery()* che restituisce una stringa, oppure si ottiene direttamente il DbCommand col metodo *BuildCommand()*.

Nel caso in cui nella stringa di ricerca fosse necessario inserire una *join* è possibile definirla nel seguente modo:

```
sqb.AddJoin(JoinType.InnerJoin, "well", "id", Comparison.Equals, "image", "wellId");
```

L'utilizzo di questo elemento è stato fondamente per produrre il codice che permettesse di creare delle *query* in modo dinamico per l'interrogazione del *database*.

Un'altra funzione che è stata impletata è il savataggio dei filtri di ricerca. Per implementarla si è modificato il codice del *framework* contenente il SelectQueryBuilder, questo non era stato scritto per essere serializzabile e in ogni classe è stato aggiunto l'attributo:

[Serializable()]

e la derivazione della classe da *ISerializable*. E' stato poi definito un nuovo oggetto: QueryObject. La sua funzione è quella di associare il nome della ricerca col relativo SelectQueryBuilder del filtro.

```
[Serializable()]
public class QueryObject : ISerializable
{
    private string name;

    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    private SelectQueryBuilder sqb;

    public SelectQueryBuilder Sqb
    {
        get { return this.sqb; }
        set { this.sqb = value; }
    }

    public QueryObject() { }

    //Costruttore per deserializzare
    public QueryObject(SerializationInfo info, StreamingContext ctxt)
    {
        this.name = (string)info.GetValue(Properties.Settings.Default.query_serialization_name, typeof(string));
        this.sqb = (SelectQueryBuilder)info.GetValue(Properties.Settings.Default.query_serialization_query, typeof(SelectQueryBuilder));
    }

    //Funzione per serializzare
    public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
    {
        info.AddValue(Properties.Settings.Default.query_serialization_name, this.name);
        info.AddValue(Properties.Settings.Default.query_serialization_query, this.sqb);
    }
}
```

Figura 4.10: Classe che definisce l'oggetto QueryObject

Prima del salvataggio della nuova ricerca, vengono caricate dal *file*, se presente, tutte le richerche già esistenti. Alla lista ottenuta viene aggiunto il QueryObject corrente e viene riscritto il file. In questo modo non ci sarà

```
//Apro il file con le query salvate per aggiungerci la query, se il file non esiste, lo creo
stream = File.Open(Properties.Settings.Default.saved_query_file, FileMode.Create);

bFormatter.Serialize(stream, listQo);
stream.Close();
```

Figura 4.11: Codice usato per la serializzazione delle ricerche su file.

```
stream = File.Open(Properties.Settings.Default.saved_query_file, FileMode.Open);
this.listQo = (List<QueryObject>)bFormatter.Deserialize(stream);
stream.Close();
```

Figura 4.12: Codice usato per la deserializzazione delle ricerche.

alcuna perdita di dati.

L'ultimo step è stato lo sviluppo dell'esportazione dei risultati. Si è deciso di rendere disponibile questa funzione per due tipi file, uno in formato Excel (xls) e l'altro in formato csv. Per implementare l'esportazione sotto file xls è stata utilizzata la libreria FileHelpers, distribuita con licenza LGPL[1] quindi utilizzabile anche per software commerciale. Questa libreria consente di creare con facilità un file Excel, chiamato Workbook nel codice, con le varie schede, Worksheet, e relative celle, Cells. (Fig. 4.13)

[1]LGPL o *Lesser General Public License* è un compromesso tra la GNU GPL e le altre licenze non *copyleft*, viene generalmente usata per le librerie *software*

```
private void CreateWorkbook(String filePath)
{
    Workbook workbook = new Workbook();
    Worksheet worksheet = new Worksheet(Properties.Settings.Default.image_save_dialog_filename);
    //riempio la prima riga col nome delle colonne
    string[] columnsName = Properties.Settings.Default.image_save_name_column.Split(',');
    for (int cn = 0; cn < columnsName.Length; cn++)
    {
        // Add column header
        worksheet.Cells[0, cn] = new Cell(columnsName[cn]);
    }
    List<string> rowItem = new List<string>();
    //campi delle righe
    if(searchedImages != null && searchedImages.Count>0){
        for(int i = 0; i < searchedImages.Count; i++){
            Image image = searchedImages[i];
            worksheet.Cells[i + 1, 0] = new Cell(image.Name);
            worksheet.Cells[i + 1, 1] = new Cell(image.Path);
            if (System.Threading.Thread.CurrentThread.CurrentCulture.ToString().Contains('i'))
            {
                string field = Properties.Settings.Default.image_save_link_it;
                string fullPath = image.Path + "\\" + image.Name;
                field = field.Replace("%P", fullPath);
                field = field.Replace("%N", image.Name);
                worksheet.Cells[i + 1, 2] = new Cell(field);
                //worksheet.Cells[i + 1, 2] = new Cell("");
            }
            else
            {
                string field = Properties.Settings.Default.image_save_link_en;
                string fullPath = image.Path + "\\" + image.Name;
                field = field.Replace("%P", fullPath);
                field = field.Replace("%N", image.Name);
                worksheet.Cells[i + 1, 2] = new Cell(field);
                //worksheet.Cells[i + 1, 2] = new Cell("");
            }
            worksheet.Cells[i + 1, 3] = new Cell(image.Date.ToShortDateString());
            if (image.Desc == null) worksheet.Cells[i + 1, 4] = new Cell("");
            else worksheet.Cells[i + 1, 4] = new Cell(image.Desc);
            if (image.ExpTime == null) worksheet.Cells[i + 1, 5] = new Cell("");
            else worksheet.Cells[i + 1, 5] = new Cell(image.ExpTime);
            worksheet.Cells[i + 1, 6] = new Cell((int)image.Gain);
            worksheet.Cells[i + 1, 7] = new Cell((int)image.MagFactor);
            worksheet.Cells[i + 1, 8] = new Cell((decimal) image.Resolution);
            worksheet.Cells[i + 1, 9] = new Cell(image.IsInsideSquare);
            worksheet.Cells[i + 1, 10] = new Cell((int)image.Width);
            worksheet.Cells[i + 1, 11] = new Cell((int)image.Height);
        }
    }
    workbook.Worksheets.Add(worksheet);

    workbook.Save(filePath);
}
```

Figura 4.13: Metodo creato per il salvataggio dei file in formato .xls

Per l'estrazione in formato csv non è stata utilizzata alcuna libreria (Fig. 4.14)

```
private void createCSV(string filePath)
{
    string delimiter = ",";
    List<string[]> rows = new List<string[]>();
    string[] columnsName = Properties.Settings.Default.image_save_name_column.Split(',');
    rows.Add(columnsName);
    if (this.searchedImages != null && this.searchedImages.Count > 0)
    {
        for (int i = 0; i < this.searchedImages.Count; i++)
        {
            Image image = searchedImages[i];
            string[] row = new string[columnsName.Length];
            row[0] = "\"" + image.Name + "\"";
            row[1] = "\"" + image.Path + "\"";
            row[2] = "\"" + image.Path + "\\" + image.Name + "\"";
            row[3] = "\"" + image.Date.ToShortDateString() + "\"";
            row[4] = "\"" + image.Desc + "\"";
            row[5] = "\"" + image.ExpTime + "\"";
            row[6] = "\"" + image.Gain.ToString() + "\"";
            row[7] = "\"" + image.MagFactor.ToString() + "\"";
            row[8] = "\"" + image.Resolution.ToString() + "\"";
            row[9] = "\"" + image.IsInsideSquare.ToString() + "\"";
            row[10] = "\"" + image.Width.ToString() + "\"";
            row[11] = "\"" + image.Height.ToString() + "\"";
            rows.Add(row);
        }
    }
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < rows.Count; i++)
    {
        sb.AppendLine(string.Join(delimiter, rows[i]));
    }
    File.WriteAllText(filePath, sb.ToString());
}
```

Figura 4.14: Metodo creato per il salvataggio dei file in formato *comma-separated value*

Entrambi i file vengono registrati col nome e nella cartella indicati dall'utente nell'apposita finestra di dialogo che viene aperta.

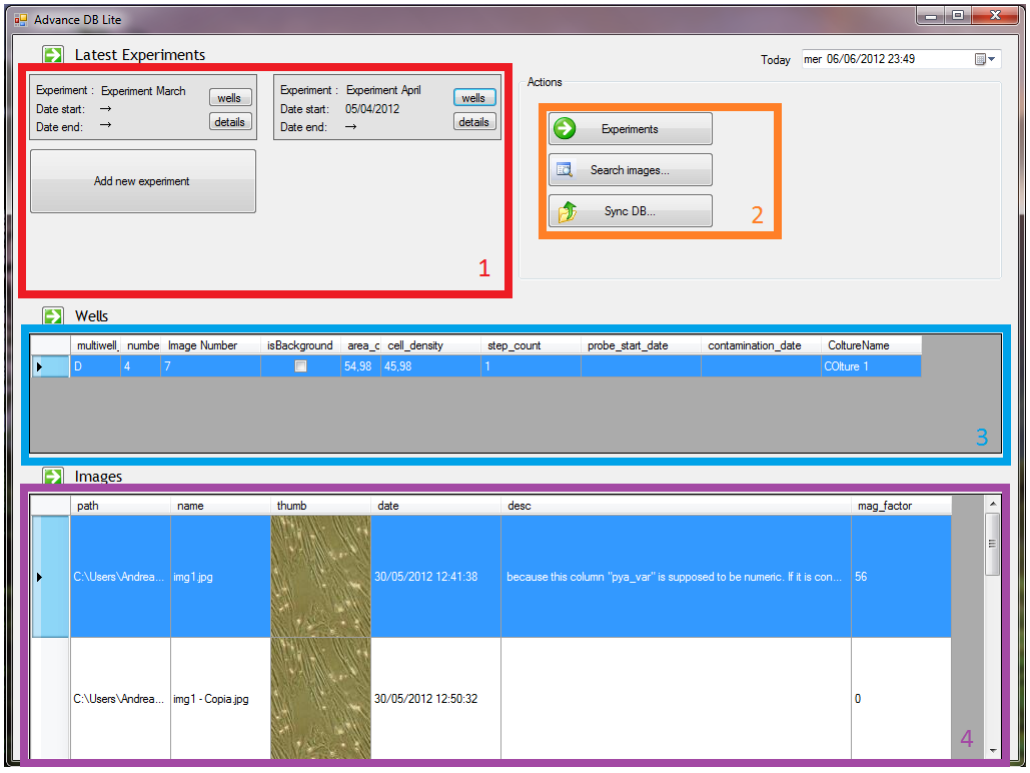
Terminata l'implementazione di tutte le funzioni di ricerca si è effettuato il *debug* dell'intero codice prodotto.

Capitolo 5

Risultati

In questo capitolo viene illustrata l'applicazione sviluppata. Saranno mostrate le varie finestre di interazione e descritto il loro utilizzo.

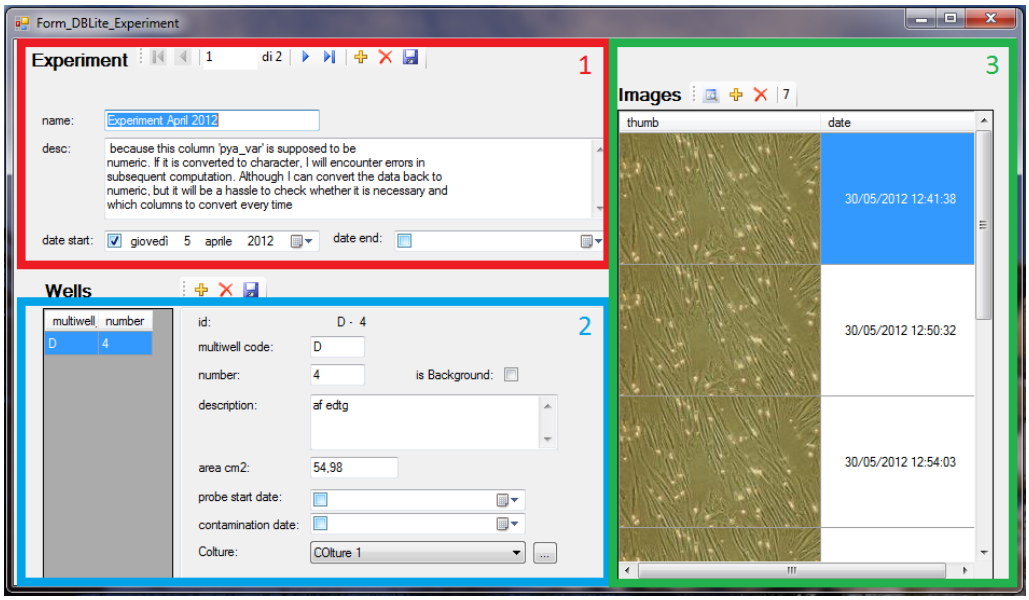
All'apertura del programma questa è la prima vista che incontriamo:



Nel riquadro (1) vengono visualizzati gli ultimi sei esperimenti creati o modificati.Nel riquadro (3) vengono visualizzati i pozzetti appartententi all'esperimento, per visualizzarli in questa tabella è sufficente cliccare sul tasto *wells* presente a fianco del nome dell'esperimento. Nel (4) sono elencate

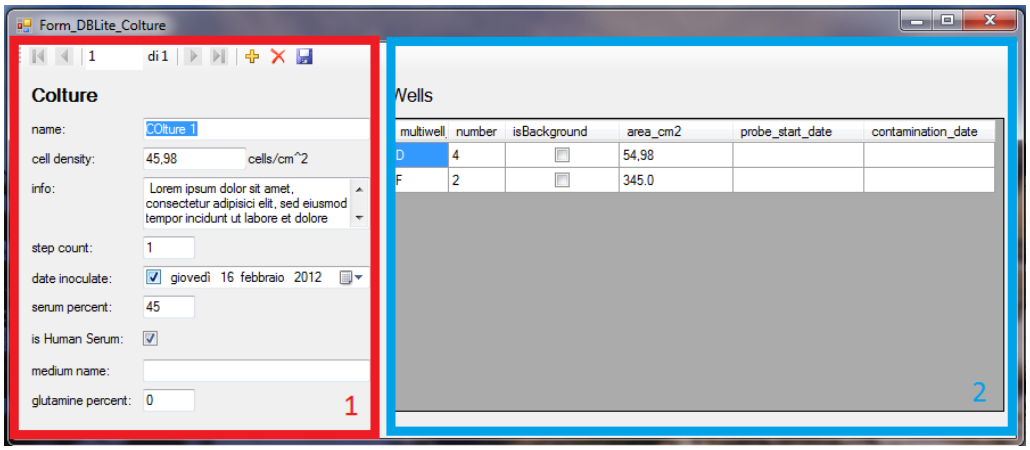
le immagini appartenenti al pozzetto selezionato nel (3). Il (2) contiene i pulsanti per visualizzare i dettagli di un'esperimento, aprire la finestra di ricerca e il comando per effettuare la sincronizzazione del *database* con una cartella presente su disco. Premendo il tasto freccia presente sopra la tabella Wells e Images viene aperta la finestra contenente i dettagli del pozzetto o dell'immagine selezionata.

La finestra che mostra i dettagli dell'esperimento è così composta:



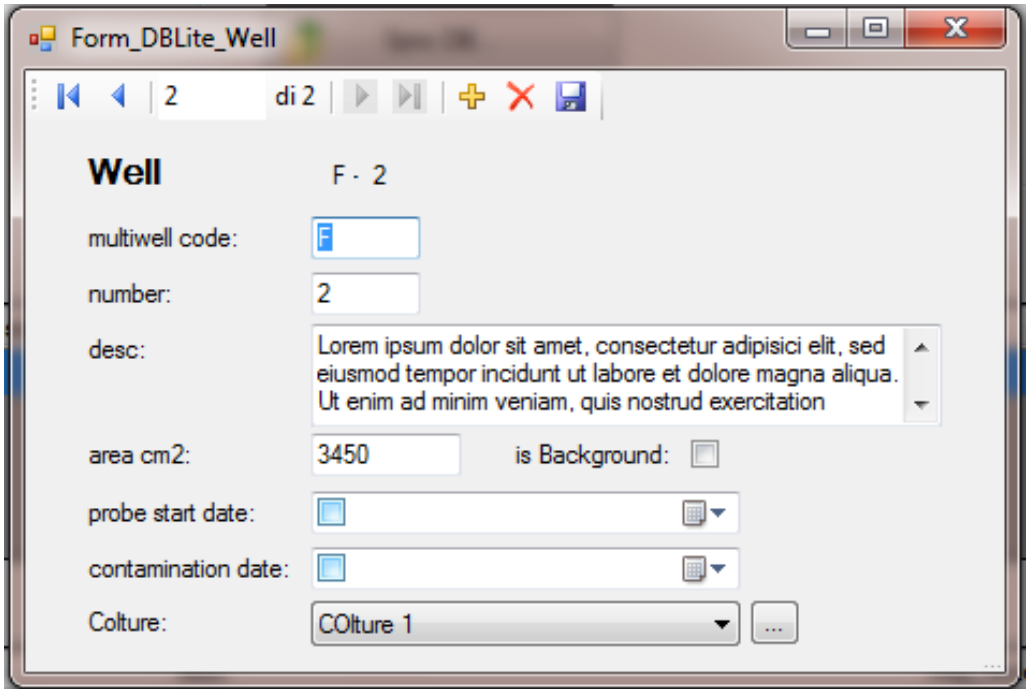
Il riquadro (1) comprende i dettagli e la gestione dell'esperimento, qui è possibile modificare, eliminare od aggiungerne uno nuovo. Attraverso il *tooltip* in alto è possibile sfogliare i vari esperimenti presenti. Nel riquadro (2) è compreso tutto ciò che riguarda le well. La tabella di sinistra le elenca, mentre nella parte destra vengono visualizzati i dettagli del pozzetto selezionato. Qui è inoltre possibile cancellarne, aggiungerne e modificarle. Premendo il tasto accanto all'elenco delle colture disponibili si accede ai dettagli. Nel riquadro(3) vengono visualizzate le immagini riferite al pozzetto selezionato.

La finestra che mostra i dettagli delle colture è così composta:



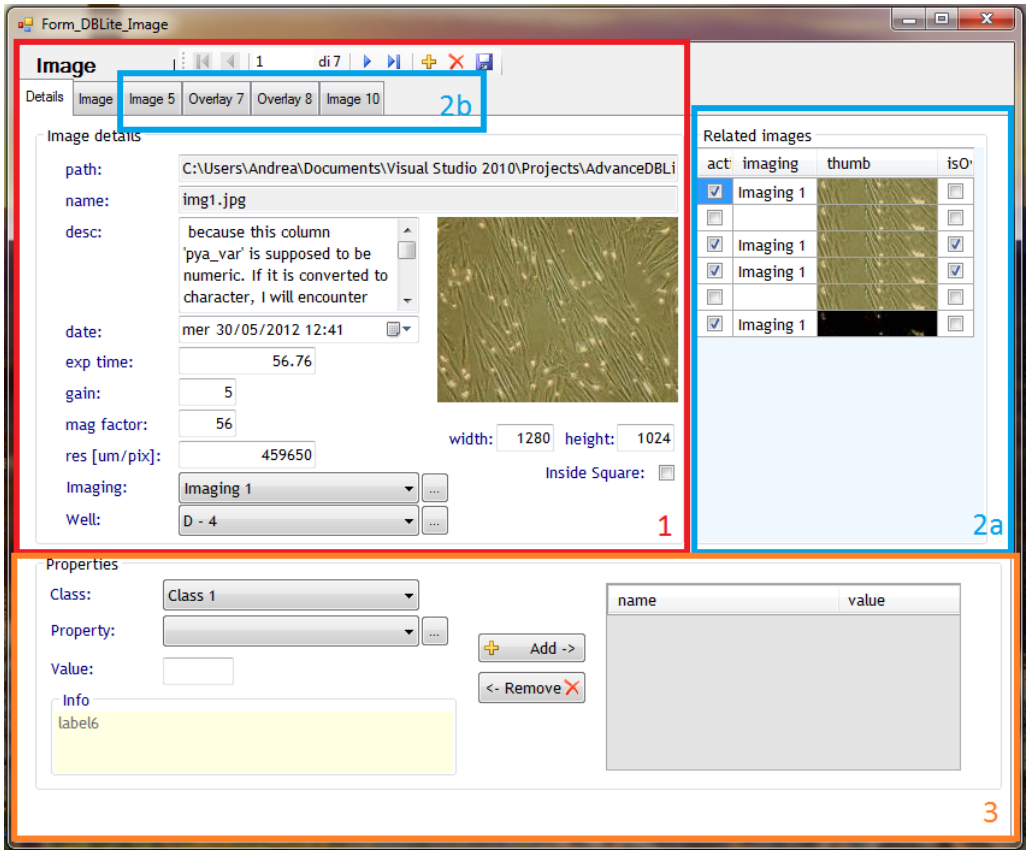
Il riquadro (1) contiene i dettagli e la gestione della coltura. Qui è possibile modificare, eliminare od aggiungerne di nuove. Attravero il *tooltip* in alto è possibile sfogliare le varie colture presenti. Nel riquadro (2) abbiamo la visuale sui pozzetti che contengono quella coltura.

La finestra che mostra i dettagli delle *well* è così composta:

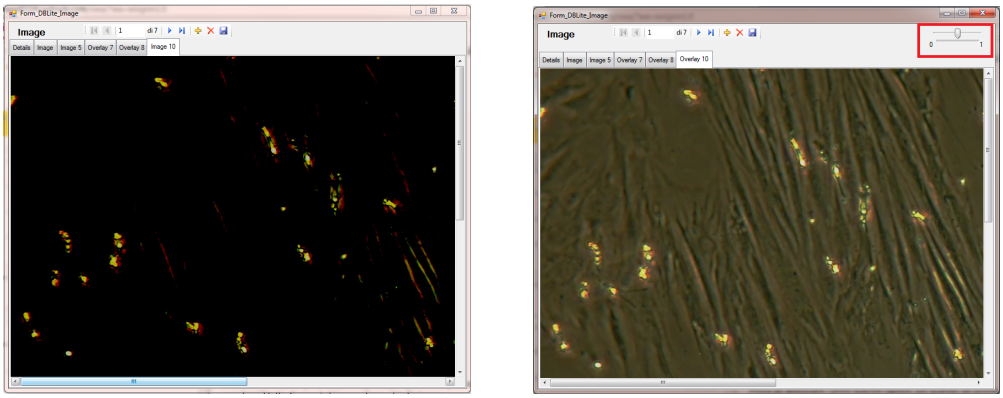


Questa finestra ci permette la visione e la gestione dei pozzetti. Col *tooltip* in alto possiamo navigare tra tutte le well comprese nell'esperimento ed è possibile modificarle, aggiungerne o eliminarle.

La finestra che mostra i dettagli delle immagini è così composta:



Il riquadro (1) contiene tutti i dettagli e la gestione dell'immagini. Dal *tooltip* presente in alto è possibile sfogliare le varie immagini appartenti al pozzetto, salvarle, cancellarle e aggiungerne. Il riquadro (2a) comprende la tabella che gestisce le immagini correlate, vengono visuallizate tutte le altre immagini appartenente al pozzetto. Attravero i *checkbox* *active* e *isOverlay* è possibile aprire nuove schede, riquadro (2b), che visualizzano la sola immagine se *isOverlay* non è *checked* o le due immagini in *alpha blending* se anche quest'ultimo è selezionato. Il riquadro (3) comprende i comandi per la valorizzazione di una proprietà dell'immagine. Cliccando nel pulsante di fianco alla tendina delle proprietà si accede al *form* per la loro gestione.

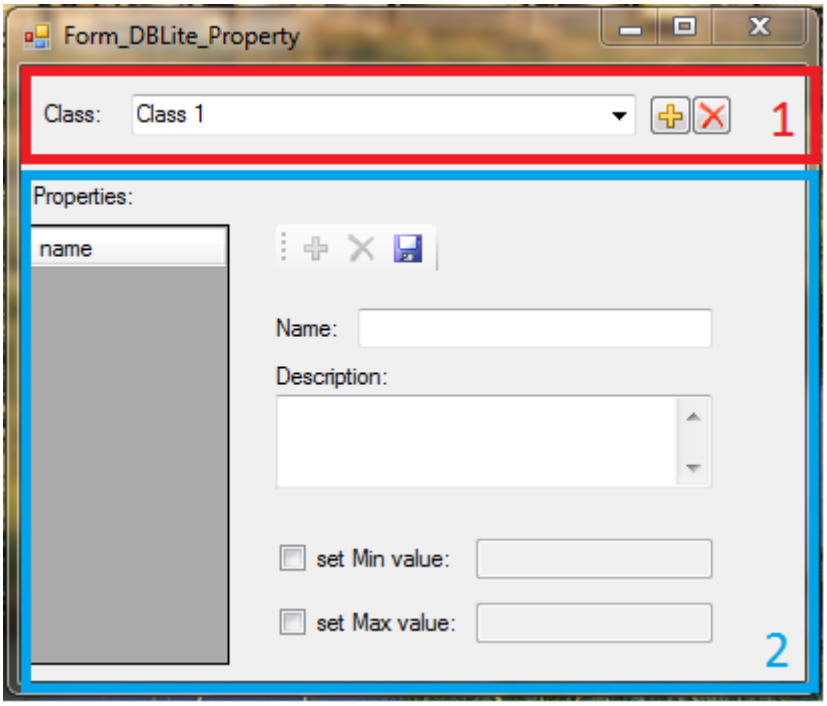


(a) Scheda contenente la sola immagine

(b) Scheda con l'*alpha blending* delle due immagini

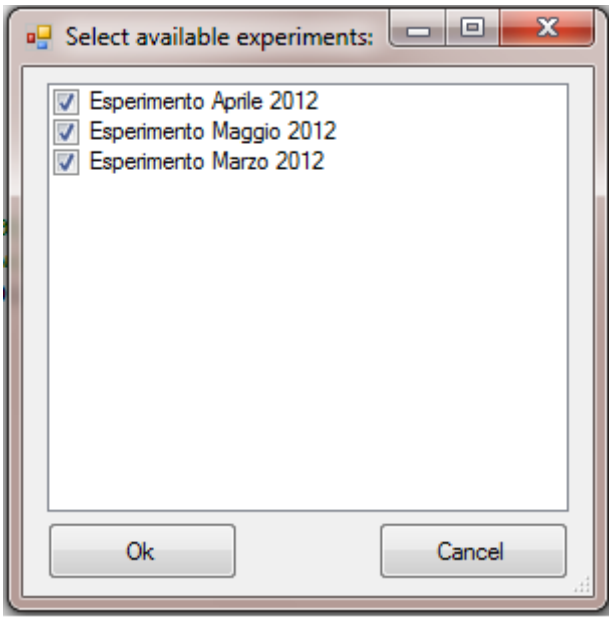
Figura 5.1: Le due schede apribili dalla tabella delle *Related Images*. In rosso è evidenziato il controllo dell'*alpha blending*

La finestra che mostra i dettagli delle proprietà è così composta:



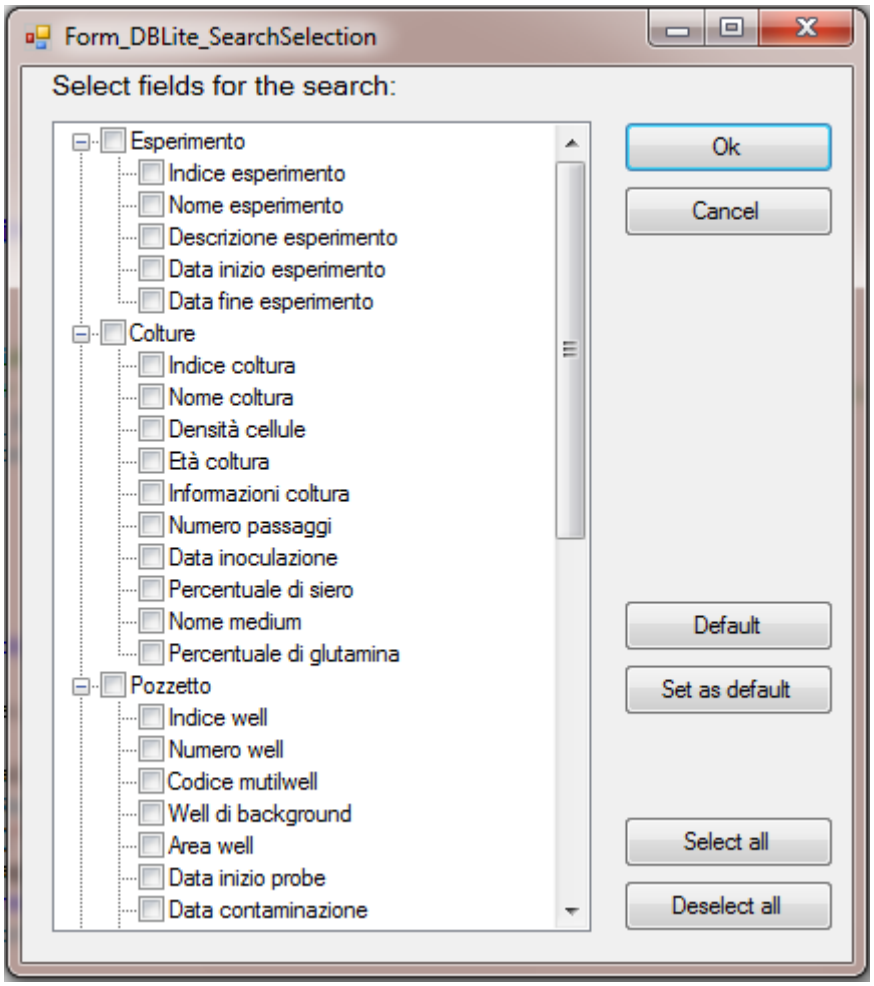
Il riquadro 1 contiene la gestione delle classi con la possibilità di aggiungerne e cancellarne. A sinistra del riquadro (2) troviamo la tabella con i nome delle varie proprietà presenti nella classe selezionata. A destra abbiamo i dettagli della classe e la possibilita di salvarla, cancellarla e aggiungerne di nuove.

Cliccando sul tasto Sync DB presente nel *landing form* si apre questa finestra:



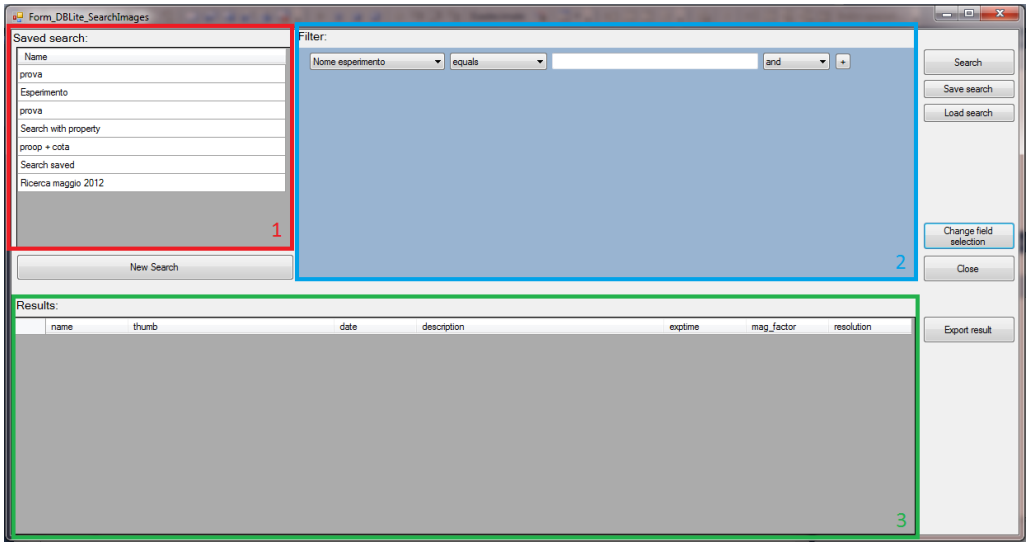
In questa finestra vi è l'elenco delle cartelle che è possibile importare sotto forma di esperimenti. Il nome dell'esperimento è il nome della cartella. Queste devono essere poste all'interno della cartella imgs all'interno dell'applicazione.

Cliccando sul tasto Search Images si apre la finestra di selezione dei campi:



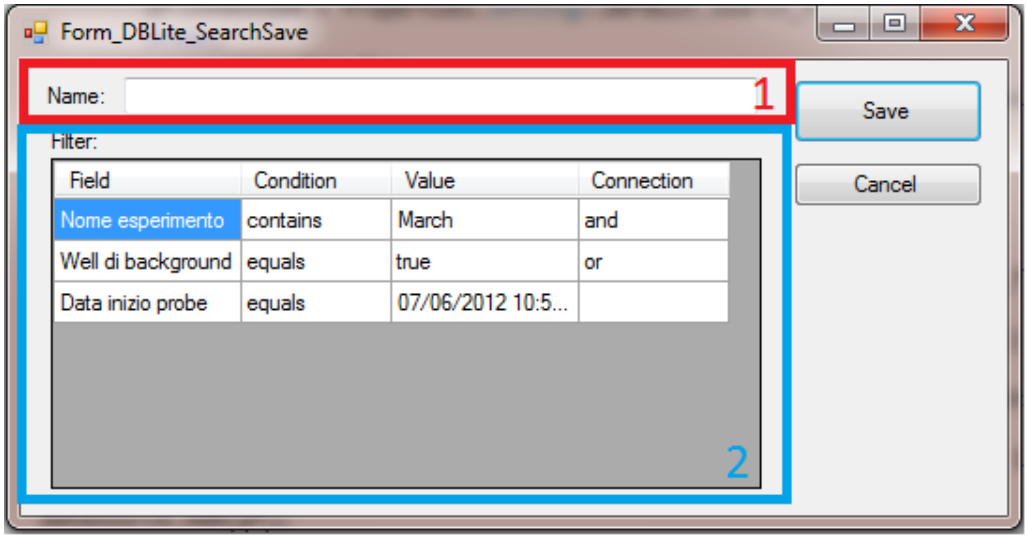
A sinistra della finestra abbiamo l'albero coi vari campi di ricerca selezionabili,selezionando il nodo di un ramo, vengono selezionati tutti i campi a lui appartenenti. Sulla destra sono posti vari controlli: In basso i pulsanti per selezionare e deselezionare tutti i campi presenti, sopra i pulsanti per settare i campi di default e per selezionarli. Cliccando su Ok si apre la finestra di ricerca.

La finestra di ricerca è così composta:



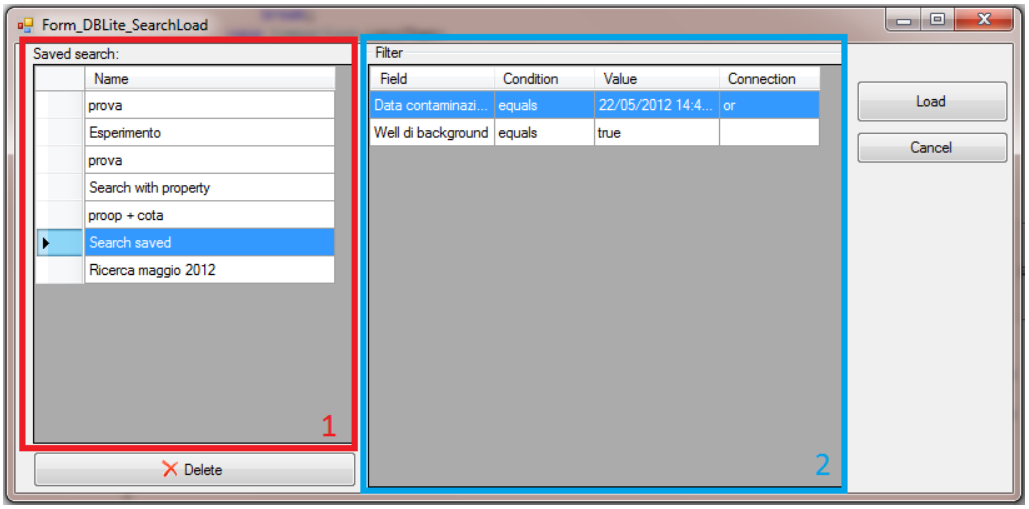
Nel riquadro (1) abbiamo l'elenco delle ricerche salvate, cliccando su una di queste verrà caricato nel riquadro (2) il filtro e nel riquadro (3) i risultati della ricerca. Cliccando sul tasto New Search si inizia una nuova ricerca, il filtro verrà resettato e i risultati della ricerca rimossi. Sulla destra abbiamo le altre funzioni: salvataggio ricerca, caricamento ricerca con visione del filtro, cambiamento dei campi selezionati e l'esportazione dei risultati.

La finestra di salvataggio di una ricerca è così composta:



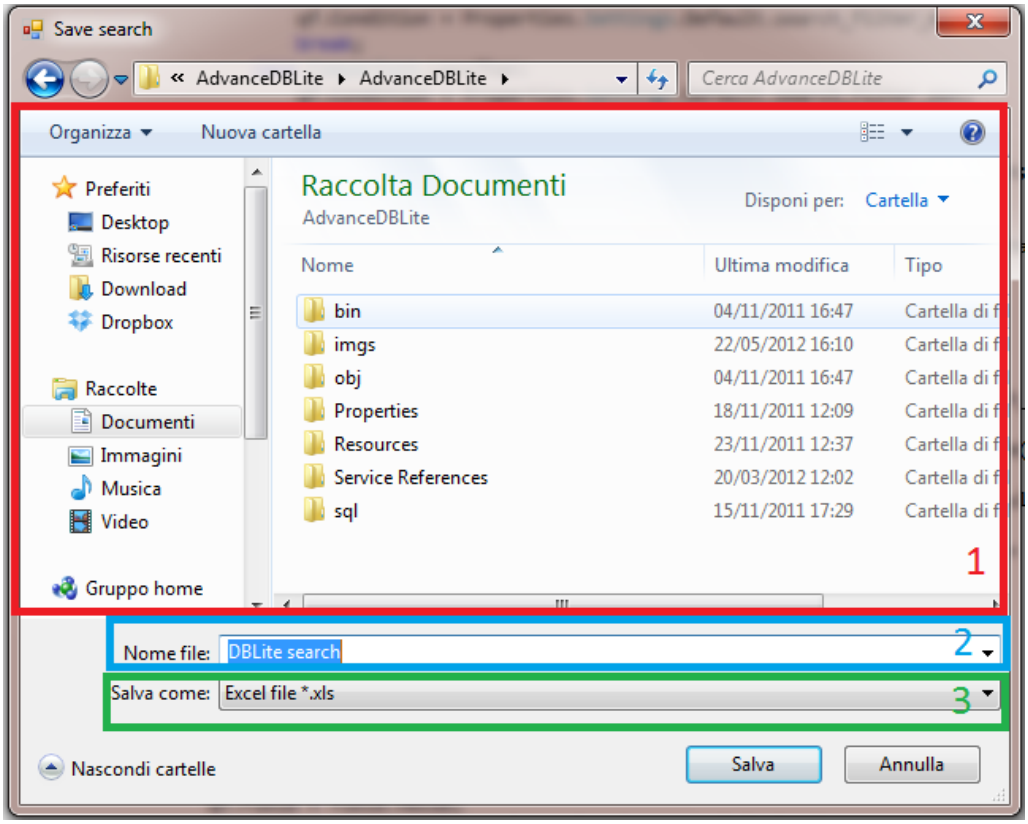
Nel riquadro (1) si inserisce il nome con cui si vuole salvare la ricerca mentre nel riquadro (2) viene visualizzato il filtro.

La finestra di caricamento di una ricerca è così composta:



Nel riquadro (1) vengono visualizzate tutte le ricerche salvate e, cliccando su una di essa, nel riquadro (2) viene visualizzato il filtro. In questa finestra è anche possibile cancellare una ricerca salvata. Cliccando sul tasto Load la ricerca viene caricata nella finestra principale.

La finestra di esportazione dei risultati è così composta:



Nel riquadro (1) è possibile selezionare la cartella in cui salvare il file, nel riquadro (2) si inserisce il nome del file e nel riquadro (3) si seleziona il tipo di file che si vuole esportare.

Il programma sviluppato è quindi in grado di gestire esperimenti di biologia cellulare effettuati in un laboratorio. Esso infatti comprende sia la parte per la catalogazione delle immaggini ricavate con le relative proprietà, sia una sezione riguardante la loro ricerca.

Capitolo 6

Conclusioni

Il lavoro svolto in questa tesi è stato rivolto all'analisi ed implementazione di un *software* per la catalogazione e ricerca di immagini acquisite durante le fasi di sperimentazione in un laboratorio di biologia cellulare. Il dominio dell'applicazione è inserito in un contesto di ricerca biologica, nel quale è necessario poter organizzare, catalogare e reperire tutte le informazioni riguardanti gli esperimenti di biologia cellulare effettuati in laboratorio.

Sebbene alcune aziende produttrici di microscopi e/o attrezzature per la cattura delle immagini, offrano già prodotti di questo tipo, questi spesso risultano avere limitazioni nel loro utilizzo, sia perché si basano sull'interfaccia proprietaria e interagiscono direttamente con lo strumento di una determinata marca, sia perché presentano un'interfaccia facente uso di una logica difficilmente interpretabile o configurabile dall'utente o, nel peggiore dei casi, non implementano tutte le funzionalità di cui il laboratorio ha bisogno.

Il lavoro qui presentato ha cercato di colmare un bisogno nato dalla necessità da parte dei biologi di operare con standard definiti dal gruppo di ricerca, implementando determinate funzioni che nascono dalla logica e dai casi d'uso dei protocolli di gestione degli esperimenti di laboratorio.

Durante la fase di analisi e progettazione si sono tenuti in considerazione diversi requisiti. In primo luogo la possibilità di poter distribuire gratuitamente il prodotto. Per poter operare una scelta su quale *database engine* utilizzare nell'applicazione, si è effettuata una analisi comparativa tra i maggiori *database* disponibili. Tra i criteri presi in esame per la scelta, condizionata anche dalle prestazione che i singoli *database* offrivano, si è tenuto conto della loro portabilità su diverse piattaforme, dall'interfacciamento

col linguaggio di sviluppo (C#) e la facilità di *porting* della struttura del database presente da MS SQL CE.

In secondo luogo l'applicazione deve rispondere a determinati standard definiti dal gruppo di utilizzo con un'interfaccia grafica il più possibile *user friendly*. La parte di catalogazione comprende l'inserimento di tutte le proprietà dell'esperimento, del pozzetto e delle immagini all'interno di un contesto ben definito: tutte le immagini appartenenti ad un determinato pozzetto, facenti parte di un dato esperimento, possono essere visualizzate insieme, così come i vari pozzetti utilizzati in un esperimento. La ricerca delle immagini in base alle proprietà testuali inserite dall'utente è stata implementata tramite un'interfaccia grafica di facile utilizzo.

Infine l'applicazione deve implementare diverse funzioni che possano migliorare e ottimizzare le fasi della ricerca bio-cellulare del laboratorio. Una di queste funzioni è rappresentata dall'*alpha blending* tra due immagini appartenenti allo stesso pozzetto: ciò permette la comparazione interattiva da parte del biologo delle stesse strutture cellulari acquisite con diverse tipologie di *imaging*, ad esempio in *brigth field* ed in fluorescenza, consentendo ai biologi di valutare sia la struttura citoplasmatica sia il conteggio dei nuclei riferiti alla stessa coltura cellulare.

Inoltre, il *software* prodotto e già predisposto per un'eventuale implementazione di una ricerca semantica per immagini. Questa predisposizione è stata possibile inserendo una parte di metadati per la gestione delle proprietà delle immagini. Attraverso un programma automatico, interfacciato col *database*, si potranno introdurre in queste proprietà dinamiche delle *features* legate alle proprietà semantiche delle immagini consentendo ad esempio funzionalità di *database image retrieval*. Questo consentirà in futuro ai biologi, attraverso l'uso di determinati algoritmi, di poter effettuare *query* per immagini.

Il lavoro in questa tesi ha consentito quindi l'introduzione di uno strumento di catalogazione e recupero delle informazioni con funzionalità mirate e di semplice utilizzo, permettendo ai biologi un miglioramento delle fasi di gestione degli esperimenti e delle attività di ricerca nel campo della biologia cellulare, ponendo anche le basi per l'implementazione di un *software* per la ricerca semantica delle immagini.

Appendice A

Licenze software

Le licenza software è uno strumento legale che spesso accompagna il prodotto, specifica le modalità con cui l'utente può utilizzare il prodotto e redistribuirlo garantendo dei diritti e imponendo obblighi. La licenza viene imposta da chi detiene il *copyright* del software, in alcuni casi il prodotto, a seconda degli usi che se ne vuole fare, può venire distribuito con licenze diverse, sarà poi l'utete ad effettuare la scelta.

L'accettazione di una licenza può avvenire in diversi modi:

- Con l'utilizzo del software, in pratica quando il software viene utilizzato si è deciso di accettare la licenza, se non lo si usa la licenza non viene accetta. Solitamente questo tipo di licenza viene adottata dai *software* che rendono disponibile anche i sorgenti oltre all'eseguibile.

- Durante l'installazione del prodotto viene chiesto esplicitamente di accettarla, in caso di risposta negativa, il *software* non verrà installato. Solitamente questo tipo di licenza viene adottato da chi distribuiscce solamente gli seguibili.

- Se il programma viene scaricato *on-line*, prima del *download* è necessario la compilazione di un form dove è obbligatorio l'accettazione della licenza.

- All'apertura della custodia del prodotto. In questo caso se la custodia non viene aperta è garantito, almeno teoricamente, la restituzione dei soldi spesi per l'acquisto.

Con la continua espansione del *software* libero o *open source* possiamo dividere le licenze in due grandi categorie: quelle rivolte alle applicazioni

proprietarie e commerciali e quelle rivolte alle applicazioni libere e *open source*.

A.1 Software proprietario

Le licenze per il *software* proprietario sono principalmente tre.

La licenza EULA o *End User License Agreement* è il contratto tra il fornitore del programma e l'utente finale. Il contenuto di questa licenza generalmente comprende la concessione di licenza, le limitazioni d'uso, garanzia e responsabilità e le restrizioni all'esportazione. Anche se slitamente è associata all'uso di applicazione proprietario, l'EULA viene utilizzata anche per accordare la licenza d'uso nei relativi termini per il *software* libero. L'accettazione della licenza avviene solitamente con la lettura del contratto presente all'interno dell'applicazione, questo comporta, in caso di non accettazione la possibilità di poter restituire il prodotto entro un determinato periodo.

La licenza *shareware* è solitamente utilizzata in quei programmi scaricabili via internet. Il *software* sotto questa licenza può essere liberamente redistribuito e può essere utilizzato per un determinato lasso di tempo (solitamente 30 giorni). Scaduti questi giorni per continuare nell'utilizzo dell'applicazione è obbligatorio per l'utente l'acquisto del prodotto.

La licenza *freeware* indica che il prodotto viene distribuito in modo gratuito. Con questo tipo di licenza il *software* può venire distribuito con o senza sorgenti ed è liberamente distribuibile e duplicabile.

A.2 Software libero o open source

Per il rilascio di programmi liberi o *open source* sono nate in questi ultimi anni moltissimei licenze, ognuna avente le proprie caratteristiche. In questo lavoro di tesi vengono illustrate le licenze con cui vengono distribuiti alcuni *database* testati.

La GNU GPL o *General Public License* è scritta dalla Free Software Fondation che ne detiene anche i diritti sul testo. Ne sono state distribuite tre versioni, l'ultima nel 2007 [14]. La versione 2.0 è la licenza di *software* libero più utilizzata al mondo. Caratteristica di questa licenza sono le possibilità

lasciate all'utente, egli infatti può modificare, copiare e redistribuire il programma, gratuitamente o a pagamento. Il produttore infatti è tenuto a rendere disponibile, anche a pagamento, il codice sorgente dell'applicazione. Rispetto alle altre licenze libere la GPL è classificabile come:

- Persistente: viene imposto un vincolo alla redistribuzione, se l'utente distribuisce copie del *software* deve farlo secondo i termini della GPL stessa.

- Propagativa: perchè l'unione di un programma coperto da GPL con un altro programma coperto da altra licenza può essere distribuito sotto GPL o non può venire distribuito. Nel primo caso si dice che la licenza è compatibile con GPL, ne secondo caso che non lo è.

La GPL quindi non permette la creazione di brevetti proporietari relativi alla modifica del *software* rilasciato con questa licenza.

La licenza LGPL o *Lesser General Public License* è scritta dalla Free Software Foundation, pubblicata nel 2007, è un compromesso tra la GNU GPL e le altre librerie non *copyleft* [15]. E' una licenza di tipo *copyleft* ma differisce dall GNU GPL perchè non richiede che eventuale *software* collegato al programma venga rilasciato sotto questa licenza, per questo motivo è principalmente usata per le librerie *software*.

La licenza IPL *Interbase Public License* e la IDPL *Initial Developers Public License* [19] sono delle varianti della MPL *Mozzilla Public License* [22]. Differiscono da questa solamente nel sottilineare il fatto che Netscape non ha creato il codice originale.

La MPL è una licenza *open source* e *free software*, è stata concepita come una versione ibrida di una licenza BSD e la GNU GPL. Questa licenza è considerata come un debole *copyleft*[1], il codice sorgente copiato o modificato sotto la licenza MPL deve rimanere sotto questa licenza. Al contrario di altre licenze, il codice sotto questa licenza può venire combianto in un programma con file proprietari che altrimenti sarebbero lavori derivata dalla MPL. Questa licenza è incompatibile con la GNU GPL.

[1]Con il termine *Copyleft* si individua un modello di gestione dei diritti d'autore basato su un sistema di licenze attraverso le quali l'autore del *software*, detentore dei diritti d'autore, indica ai fruitori dell'opera che essa può essere utilizzata, diffusa e modificata, nel rispetto delle condizioni espresse dalla licenza.

La *PostgreSQL License* è una licenza simile alla BSD [17]. Il codice rilasciato sotto questa licenza può essere usato, modificato e distibuito per qualsiasi scopo.

Le licenze BSD sono una famiglia di licenze permissive per *free software* [27]. Il nome deriva dalla licenza originale usata per la distribuzione del sistema operativo Unix *Berkeley Software Distribution* (BSD). Il *software* rilasciato sotto questa licenza è liberamente modificabile e ridistribuibile. La modifica di un programma con licenza BSD può essere ridistribuito usando la stessa o qualunque altra licenza senza dover rendere pubblico il codice sorgente delle modifiche.

Bibliografia

[1] IBM Corporation 1994. *About DB2 Express-C*. URL: http://www-01.ibm.com/software/data/db2/express/about.html.

[2] Peter J. Brown. «Software portability». In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley e Sons Ltd., pp. 1633–1634. ISBN: 0-470-86412-5. URL: http://dl.acm.org/citation.cfm?id=1074100.1074809.

[3] VERSANT CORP. *Open Source Object Database (OODB) :: Product Information*. URL: http://www.db4o.com/about/productinformation/.

[4] E. F. Codd. «A relational model of data for large shared data banks». In: *Commun. ACM* 13.6 (giu. 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[5] E. F. Codd. «Further Normalization of the Data Base Relational Model». In: *ACM Transactions on Database Systems* (1971).

[6] Actian Corporation. *Ingres overview*. URL: http://www.actian.com/products/ingres/overview.

[7] Oracle Corporation. *MySQL Technical Specifications*. URL: http://www.mysql.it/products/enterprise/techspec.html.

[8] *Datatypes In SQLite Version 3*. URL: http://www.sqlite.org/datatype3.html.

[9] ECMA. *Standard ECMA-334 C Sharp Language Specification*. quarta. 2006.

[10] Klaus Elhardt e Rudolf Bayer. «A database cache for high performance and fast restart in database systems». In: *ACM Trans. Database Syst.* 9.4 (dic. 1984), pp. 503–525. ISSN: 0362-5915. DOI: 10.1145/1994.1995. URL: http://doi.acm.org/10.1145/1994.1995.

[11] *Features Of SQLite*. URL: http://www.sqlite.org/features.html.

[12] *File Locking And Concurrency In SQLite Version 3*. URL: http://www.sqlite.org/lockingv3.html.

[13] Myron Flickner et al. «Query by Image and Video Content: The QBIC System». In: *Computer* 28 (1995), pp. 23–32. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/2.410146.

[14] Free Software Foundation. *GNU GENERAL PUBLIC LICENSE*. 2007. URL: http://www.gnu.org/copyleft/gpl.html.

[15] Free Software Foundation. *GNU LESSER GENERAL PUBLIC LICENSE*. 2007. URL: http://www.gnu.org/licenses/lgpl.html.

[16] L. Guida F. Mattioli S. Penco P. Romano L. Scarabelli G. Luigi Mariottini V. Capicchioni. *Introduzione alle colture cellulari*. A cura di Escom. Tecniche Nuove, 2010. ISBN: 884812433X.

[17] PostgreSQL Global Development Group. *PostgreSQL Database Management System*. URL: http://wiki.postgresql.org/wiki/FAQ#What_is_the_license_of_PostgreSQL.3F.

[18] The PostgreSQL Global Development Group. *Postgres SQL Wiki*. URL: http://wiki.postgresql.org/wiki/Main_Page.

[19] *Initial Developer's PUBLIC LICENSE*. URL: http://www.calculate-linux.org/packages/licenses/IDPL.

[20] Michael J. Kamfonas. «Recursive Hierarchies: The Relational Taboo!» In: *The Relational Journal* (1992).

[21] Won Kim. *Introduction to Object-Oriented Databases*. A cura di Mit Pr. 1990. ISBN: 0-262-51216-5.

[22] *Mozilla Public License*. URL: http://www.mozilla.org/MPL/2.0/.

[23] Eric J. Simon Neil A. Campbell Jane B. Reece. *L'essenziale di biologia*. A cura di Pearson. 2008. ISBN: 8871923995.

[24] Firebird Project. *Firebird Release Note*. URL: http://www.firebirdsql.org/en/release-notes/.

[25] *Sharp-SQL DBMS*. URL: http://sharpsql.sourceforge.net/.

[26] Antonio Teti. *Il Futuro Dell'information e Communication Technology*. A cura di Springer Verlag. 2009. ISBN: 8847013879.

[27] *The BSD License*. URL: http://www.opensource.org/licenses/bsd-license.php.

Ringraziamenti

Colgo l'occasione per ringraziare tutti coloro che mi hanno permesso di effettuare questo periodo formativo, la mia famiglia che mi ha sempre supportato e sopportato e tutti i miei amici.

Un ringraziamento particolare al professore Alessandro Bevilacqua che mi ha dato l'opportunità di poter svolgere questo lavoro di tesi e alll'Ing. Alessandro Gherardi che mi ha seguito costantemente durante tutto il lavoro.