Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

Analisi e utilizzo di DBMS per Android su dispositivi mobili

Laureando:
Filippo Baccaglini

Relatore:
Prof. Giorgio Maria Di Nunzio

Anno Accademico 2011/2012

Indice

Elenco delle figure

Elenco dei listati codice

Capitolo 1

Introduzione

Lo scopo di questa tesi è valutare le possibilità offerte dal *DataBase Management System (DBMS)* SQLite per Android, sotto il punto di vista della gestione di basi di dati, siano esse grandi o piccole, complesse o meno.
Si è deciso di intraprendere questa strada in quanto al giorno d'oggi Android è uno dei sistemi operativi per dispositivi embedded più diffusi e offre numerose applicazioni e potenzialità in molti campi grazie alla sua flessibilità, essendo, tra le altre caratteristiche, programmabile in Java.
In un primo momento si è effettuata una ricerca di mercato per capire la situazione attuale del consumo di prodotti basati sulla gestione di basi di dati, qual è la complessità dei database e quali sono gli scopi per cui sono stati creati. Poi si è cercato di scrivere due applicazioni che possano offrire un approfondimento sulle possibilità di SQLite su piattaforma Android.
La prima cerca di coprire una parte di mercato che non sembra essere ancora inflazionata, ovvero la creazione di un gestionale per piccole aziende, che implica la realizzazione di un database di dimensioni e complessità medio-piccole.
La seconda invece si porta ad un livello più accademico, ovvero la gestione di grandi basi di dati di complessità medio-alta.
Il tutto è stato realizzato avendo già a disposizione del codice SQL scritto per altri DBMS, per cui si sono valutati il grado di compatibilità di SQLite con altri sistemi di gestione di basi di dati e le tecniche per incrementarla.
Si cercherà infine di capire quali sono i limiti sia dell'ambiente di sviluppo, sia delle reali applicazioni di SQLite su piattaforma Android.

Capitolo 2

Analisi del mercato

L'analisi di mercato è stata effettuata presso l'Android Market [1], usando varie chiavi di ricerca. Quelle con risultati più rilevanti sono state *database*, *dbms* e *SQLite*.
Di seguito vengono riportate le applicazioni di maggior interesse ai fini dell'analisi, con la propria descrizione riassunta da quella presente sul Market ed un commento personale dopo aver effettuato delle prove.

2.1 Memento Database

Descrizione:

Memento. Your android remembers everything[2].
Memento è un database personale che permette di immagazzinare tutti i propri dati (collezioni, acquisti, inventario, ricette, impegni, pagamenti) in un unico posto.
Caratteristiche principali:

- memorizzazione voci con campi personalizzati;
- ordinamento, raggruppamento e filtraggio delle voci per qualunque campo;
- 19 tipi di campi: testo, interi, booleani, data/tempo, percentuale, valuta, immagine, audio, contatto, coordinate *Google Maps* e altri;
- sincronizzazione con il servizio *Google Docs*;
- importazione ed esportazione in formato *CSV*;
- protezione con password (voci criptate con *AES-128*).
- backup e ripristino dei dati;
- invio delle voci via SMS, e-mail e altri servizi disponibili;

[1] raggiungibile al sito `https://market.android.com`
[2] Memento Database `http://luckydroid.com/`

- scanner codici a barre, creazione di voci con dati e immagini ottenute dal servizio *Google Base*;

Commento:

Memento Database permette di racchiudere in un'unica applicazione tutti i propri dati personali, da una lista di cd alle proprie password, alle note prese durante una lezione. Interessante come idea ma non si sa se e come venga utilizzato il DBMS *SQLite*, essendo l'utente completamente all'oscuro dell'organizzazione dei dati, mascherata dall'interfaccia utente. Applicazione gratuita.

2.2 HanDBase Database Manager

Descrizione:

HanDBase is a Powerful Relational Database Manager for Mobile Devices[3]. Gestore di database relazionale, le cui caratteristiche principali sono:

- creare e modificare database relazionali direttamente sul proprio telefono;
- 19 tipi di campo;
- ordinamento, filtraggio e viste personalizzate;
- criptazione dei campi;
- più di 2000 modelli disponibili;

La versione desktop e l'applicazione per la sincronizzazione sono vendute separatamente.

Commento:

Dalla descrizione sembra possibile che l'applicazione utilizzi *SQLite*, anche se non è ben chiaro il meccanismo di traduzione tra vari RDBMS che possono essere installati sulle postazioni desktop o server. Prezzo al di sotto di 10 euro.

[3]HanDBase Database Manager `http://www.ddhsoftware.com/handbaseandroid.html`

2.3 Cellica Database for Android

Descrizione:

Con questa applicazione è possibile leggere e scrivere su qualsiasi database lato desktop in modalità wireless con il proprio dispositivo Android (3G, GPRS, EDGE o Wi-Fi). Le modifiche apportate da uno o l'altro lato possono essere sincronizzate, tramite l'apposito software desktop. Caratteristiche principali:

- sincronizzare qualsiasi database, viste e procedure lato desktop in modalità wireless su dispositivi mobili Android (via 3G, 4G, EDGE o Wi-Fi) e viceversa;
- effettuare query SQL SELECT, filtri e ordinare campi;
- basi di dati supportate: Microsoft Access, Access 2007, Microsoft Excel, Excel 2007, Oracle, SQL Server, DB2, MySQL, PostgreSQL, FoxPro, dBase, R:base e qualsiasi database ODBC;
- creare database personalizzati su dispositivo Android, o scegliere tra uno dei modelli predefiniti come VehicleInfo, BankAccount, MembershipInfo, ecc.
- dati crittografati con AES a 128 bit;
- supporto database Unicode in lingue come giapponese, cinese, coreano, russo, ecc.;
- moduli supportati su dispositivi Android;
- strutturare la maschera nell'applicazione desktop con etichette, campo di testo, pulsanti, checkbox, casella combinata, pagina, controllo sub-form;
- supporta sotto-moduli e la progettazione di un form Master/Details, o di form con struttura genitore/figlio utilizzando subform;
- invio di dati su singolo record di modulo come un'immagine PNG via e-mail;
- crea una maschera in verticale / orizzontale su dispositivo mobile;
- importazione ed esportazione di progetti di form;

La componente desktop dell'applicazione funziona solo su PC-Windows ed è disponibile sul sito dello sviluppatore [4] [5] Esiste anche un'altra versione di questo software, che permette l'utilizzo all'interno di una LAN tramite Wi-Fi, le cui caratteristiche aggiuntive sono:

- supporto a firma di controllo;

[4] Applicazione: `http://www.cellica.com/download/CellicaDatabaseAndroid.exe`
[5] Guida per l'utente: `http://www.cellica.com/AndroidCellicaDatabaseUserGuide.pdf`

- sincronizzazione immediata dei dati in entrambe le direzioni, comprese le immagini con il database lato desktop;
- non necessita di collegamento Internet sul dispositivo. La versione Internet (descritta precedentemente) permette di sincronizzare i dati in mobilità, mentre la versione Wi-Fi supporta la sincronizzazione di dati solo in rete locale;

Commento:

Essendo l'unica applicazione trovata sul Market che permette di dare portabilità a database presenti su postazioni fisse, si rende estremamente utile nelle operazioni di inventariazione nei magazzini. Le lacune sono nel fatto che è possibile interfacciarsi solamente a un PC-Windows e la mancanza di lettori codici a barre e QRCode, che avrebbero potenziato enormemente l'applicazione. Il software lato desktop è disponibile sul sito degli sviluppatori inserendo un codice misto IMEI/MEID/MAC che appare sul display del dispositivo Android all'avvio dell'applicazione.
Questa applicazione è gratuita per un periodo di valutazione di 10 giorni, mentre la versione completa è disponibile solamente dal sito web degli sviluppatori con un prezzo intorno ai 40 dollari [6].

2.4 MySQL client per Android

Descrizione:

MySql per Android[7] consente di connettersi a un database remoto MySQL già esistente.
Le caratteristiche principali sono:

- selezionare, inserire e aggiornare i dati;
- utilizzare un menu cronologia delle query;
- esportare i risultati in un file CSV;
- supporta sia la visualizzazione orizzontale (*landscape*) con scorrimento, che quella verticale (*portrait*);

Commento:

Creando l'applicazione una connessione remota ad un DBMS, non si rende necessaria la creazione di un database locale, ma solamente un'interfaccia utente. Prezzo al di sotto di 5 euro.

[6] Versione completa di Cellica Database: `http://www.cellica.com/cellicadatabaseandroid.html`

[7] MySQL client per Android `http://www.akebulan.com/mysql_android/mysql_android.html`

2.5 SQLite Editor

Descrizione:

SQLite Editor[8], per utenti con permessi di *root*, lista tutte le applicazioni installate che hanno un database locale interno al dispositivo. È possibile quindi selezionare un'applicazione e modificarne il database. Permette l'integrazione con altre applicazioni preposte all'esplorazione dei files all'interno del dispositivo. Per utenti senza permessi di *root*, è possibile visualizzare e modificare database solo sull'*SD card*. I dati sono visualizzati in tabelle con scorrimento in orizzontale ed è possibile applicare filtri su ogni campo.

Commento:

Questo software permette la visualizzazione dei record in una tabella, oltre all'aggiunta, modifica o rimozione di tuple da relazioni esistenti, ma non offre nessun'altra funzione. Applicazione disponibile ad un prezzo inferiore ai 5 euro.

2.6 SQLite Manager

Descrizione:

SQLite Manager[9] è un'applicazione per la visualizzazione di database SQLite interni al dispositivo mobile, le cui caratteristiche principali sono:

- file manager integrato, per localizzare database SQLite;
- visualizzazione di tabelle di database SQL;

Commento:

Quest'applicazione permette di visualizzare relazioni in una tabella, di effettuare query per la modifica della struttura dei dati e per effettuare interrogazioni. L'interfaccia utente però è poco comprensibile e immediata. Software disponibile gratuitamente o in versione a pagamento con qualche funzionalità aggiuntiva.

[8]SQLite Editor, e-mail dello sviluppatore `support@speedsoftware.co.uk`
[9]SQLite Manager `http://xuecs.com/blog/`

2.7 aSQLiteManager

Descrizione:

aSQLiteManger[10] è un gestore di database SQLite, le cui caratteristiche principali sono:

- modifica sicura di tutti i database per utenti con permessi di *root* attraverso *aShell* (caratteristica in versione *beta*);
- apertura di database all'interno dell'applicazione;
- integrazione con altri file manager per l'apertura di database;
- visualizzazione di relazioni, dati, strutture e definizioni in SQL per tabelle e viste e creazione di viste ed indici;
- esecuzione di qualunque comando SQL e di script SQL anche a più righe, con esportazione dei risultati in file ASCII;
- aggiunta e modifica di tuple attraverso l'interfaccia;
- copia del contenuto di ogni cella negli appunti;
- memorizzazione di tutti i comandi eseguiti nei database aperti;
- creazione di transazioni, con *BEGIN*, *COMMIT* e *ROLLBACK*;
- esportazione e ripristino di database in e da script SQL;

SQLite non verifica i tipi di dato durante l'inserimento, ma aSQLiteManager esegue i controlli al momento dell'inserimento nell'interfaccia, rispecchiando i tipi di dato definiti nelle relazioni.

Commento:

Questo software permette di visualizzare viste e indici, di effettuare query (sia per modifica della struttura dei dati sia per l'accesso a informazioni) e di creare transazioni. È possibile infine creare un nuovo database. Applicazione gratuita.

[10] aSQLiteManager `http://aaa.andsen.dk/aSQLiteManager.html`

2.8 RL Benchmark: SQLite

Descrizione:

RL Benchmark: SQLite[11] effettua dei test per determinare i tempi utilizzati dal proprio dispositivo per processare un buon numero di query SQL.

Commento:

Questo software effettua un benchmark del DBMS sul proprio dispositivo. La valutazione è effettuata su:

- 1000 inserimenti, 25000 inserimenti su transazione, 25000 inserimenti su transazione in tabelle indicizzate, inserimenti da selezioni;
- 100 selezioni senza indice, 100 selezioni con comparazione di stringhe, 5000 selezioni con indice;
- creazione di un indice,
- 1000 aggiornamenti senza indice, 25000 aggiornamenti con indice;
- cancellazione con e senza indice, cancellazione di tabella

L'applicazione al termine dei test riporta i propri risultati e permette il confronto con dati di altri dispositivi, anche se un po' datati. Software disponibile gratuitamente su Android Market.

2.9 Conclusioni

Dalla ricerca su Android Market di applicazioni che utilizzino il DBMS SQLite, includendo anche i risultati non analizzati in particolare, è emerso che è comune uno sfruttamento solo marginale del database, per l'immagazzinamento e la consultazione di dati personali, di tabelle per *datasheet* di componenti elettronici, di coordinate geografiche (autovelox, POI, ecc.), liste della spesa, ecc.
Gli esempi trovati in Internet o in letteratura per implementare applicazioni che sfruttino il DBMS in questione, evidenziano una comune mancanza di profondità e complessità dei database creati, in quanto sembra estremamente diffusa la pratica di creare basi di dati costituite da una sola relazione e con pochissimi vincoli, se non addirittura nessuno. In casi più rari, si è notata la creazione di database con più relazioni, ma ancora con pochi o nessun vincolo e senza utilizzo di chiavi esterne.
In questa tesi si andranno perciò a testare ed analizzare le capacità del DBMS SQLite integrato in Android utilizzato come vero e proprio *DataBase Management System*, creando basi di dati adatte a gestire grandi moli di dati o per un utilizzo specifico.

[11] RL Benchmark: SQLite `http://redlicense.com/`

Capitolo 3

Progettazione e sviluppo di database

In questo capitolo si andranno ad analizzare la progettazione e lo sviluppo di due database: uno di media dimensione e complessità, creato per gestire contatti e lavorazioni in una piccola azienda, uno invece adatto a gestire esperimenti di *information retrieval*.
A questo scopo si è provveduto ad installare sulla macchina utilizzata per la creazione dei software l'apposito IDE per lo sviluppo di applicazioni su piattaforma Android.

3.1 Installazione dell'ambiente di sviluppo integrato

Per installare l'IDE è stata seguita la guida presente sul sito preposto allo sviluppo di software su piattaforma Android[1], che verrà di seguito riassunta.

3.1.1 Fase 1 - Preparazione dell'ambiente

Per poter sviluppare applicazioni su piattaforma Android, è necessario prima di tutto avere installato sulla propria macchina *JDK 5* o *JDK 6*[2](*Java Development Kit*), oltre all'ambiente di programmazione *Eclipse 3.6*[3] (Helios) o superiore, di cui è raccomandato l'uso della versione *Eclipse classic*.

[1]Guida all'installazione: http://developer.android.com/sdk/installing.html
[2]Download JDK: http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3]Download Eclipse: http://www.eclipse.org/downloads/

3.1.2 Fase 2 - Installazione dell'SDK starter package

L'SDK starter package, disponibile sul sito[4], include solamente gli strumenti di base per lo sviluppo, ma rende possibile il download di ulteriori componenti, quali ad esempio librerie per nuove versioni di Android, aggiornamenti agli strumenti di sviluppo, strumenti per sviluppare su dispositivi particolari.
È consigliabile, dopo l'installazione, aggiungere i percorsi alle variabili d'ambiente del sistema operativo per un utilizzo più immediato.

3.1.3 Fase 3 - Installazione del plugin ADT su Eclipse

Per facilitare enormemente lo sviluppo di applicazioni, è stato creato un plugin per l'IDE Eclipse, chiamato appunto *ADT* (Android Development Tools). L'installazione avviene aggiungendo ai *repository* di Eclipse l'URL **https://dl-ssl.google.com/android/eclipse/** e installando i componenti proposti. Sucessivamente è necessario impostare, nella sezione *Android* delle preferenze di Eclipse, la posizione del SDK installato in precedenza.

3.1.4 Fase 4 - Aggiunta di piattaforme e altri componenti

Per poter sviluppare un applicazione, è necessario scaricare dall'SDK i componenti essenziali alla creazione di software, quali piattaforme, strumenti, add-ons, esempi e documentazione.

3.1.5 Fase 5 - Operazioni finali ed indicazioni

Svolte tutte e 4 le fasi, non resta che creare un dispositivo virtuale per poter effettuare test preliminari sulla propria applicazione, operazione possibile all'interno del software Android SDK. É necessario ricordare che l'emulatore purtroppo non funziona correttamente quando si tratta di simulare flussi audio/video in entrata, o se si tratta di utilizzare sensori.
Lo strumento principale di cui si farà uso, soprattutto nelle prime fasi di sviluppo del software, è il *LogCat*, eseguibile dalla shell del comando *adb* (Android Debug Bridge). LogCat permette di avere su terminale o all'interno di Eclipse (per una migliore visualizzazione), un output di debug di tutto il sistema, di ogni operazione svolta dalla Virtual Machine e di ogni errore che si presenta, con la propria descrizione, come se si trattasse di un normale software scritto in Java.

[4]Download SDK starter package: http://developer.android.com/sdk/index.html

3.2 Tipi di dato in SQLite

La documentazione completa è disponibile sul sito di SQLite[5].
Ogni dato immagazzinato o manipolato in un database SQLite fa parte di una tra le *classi di immagazzinamento* NULL, INTEGER (fino a 8 byte con segno), REAL (valori a virgola mobile a 8 byte), TEXT e BLOB.
Si parla appunto di classi di immagazzinamento perché sono molto più generiche dei tipi di dato (ad esempio la classe INTEGER include 6 tipi di dato integer a 1, 2, 3, 4, 6 e 8 byte). In SQLite 3, tutte le colonne tranne quella definita come *INTEGER PRIMARY KEY* possono contenere dati di qualunque classe di immagazzinamento.
SQLite non possiede né un tipo di dato booleano, per il quale si usano gli interi 0 (falso) e 1 (vero), né un tipo di dato per le date e il tempo, per le quali si hanno tre possibilità: salvarle come TEXT (come stringhe ISO8601 YYYY-MM-DD HH:MM:SS.SSS), come REAL (come numero di giorni del calendario Giuliano a partire dal mezzogiorno a Greenwich del 24 Novembre 4714 A.C., in accordo con il calendario Gregoriano prolettico) o come INTEGER (come Unix Time, il numero di secondi dal 01/01/1970 alle 00:00:00 UTC). Le applicazioni possono scegliere come salvare date e tempi liberamente, convertendole da un formato all'altro con le funzioni interne a SQLite.

Affinità dei tipi di dato

Per massimizzare la compatibilità tra SQLite ed altri DBMS, viene introdotto il concetto di affinità dei tipi di dato, ovvero il tipo di dato raccomandato (ma non richiesto) per i dati di una colonna di una tabella, che può comunque contenere qualsiasi tipo di dato.
La classe di immagazzinamento preferita per una colonna è chiamata appunto *affinità*. Tali affinità possono essere riferite alle classi TEXT, NUMERIC, INTEGER, REAL e NONE (per nessuna affinità).
Ad esempio una colonna con affinità alla classe TEXT immagazzina dati con le classi di immagazzinamento NULL, TEXT o BLOB; se viene inserito un dato numerico in una colonna con affinità alla classe TEXT, viene convertito in testo prima di essere inserito.
Una colonna con affinità alla classe NUMERIC può contenere valori utilizzando tutte e cinque le classi di immagazzinamento: quando un testo viene inserito in una colonna NUMERIC, la classe di immagazzinamento del testo viene convertita in INTEGER o REAL (in ordine di preferenza) se tale conversione non comporta perdita di dati (*lossless*) ed è reversibile.
Per le conversioni tra le classi di immagazzinamento TEXT e REAL, SQLite considera la conversione lossless e reversibile se le prime 15 cifre decimali significative del numero sono preservate. Se la conversione lossless da TEXT a INTEGER o REAL non è possibile allora il dato è salvato utilizzando la classe di inserimento TEXT.

[5] http://www.sqlite.org/datatype3.html

Nessun tentativo di conversione viene effettuato per convertire valori BLOB o NULL.
Se una stringa assomiglia ad un letterale a virgola mobile con un punto decimale e/o una notazione esponenziale, ma può essere espressa come INTEGER, l'affinità NUMERIC la convertirà in un intero (ad esempio, la stringa '3.0e+5' è inserita in una colonna con affinità NUMERIC come l'intero 300,000, non come il valore in virgola mobile 300,000.0).
Una colonna con affinità INTEGER si comporta come una colonna con affinità NUMERIC; l'unica differenza è evidente nelle espressioni di *CAST*.
Una colonna con affinità REAL si comporta come una colonna con affinità NUMERIC, tranne per il fatto che forza gli interi ad essere rappresentati in virgola mobile. Come ottimizzazione interna, i numeri a virgola mobile piccoli senza parte frazionaria inseriti in colonne con affinità REAL, sono scritti sul file system come interi per risparmiare spazio, ma sono automaticamente riconvertiti in virgola mobile al momento della lettura; tale ottimizzazione è comunque invisibile a livello SQL.
Una colonna con affinità NONE non ha classi di immagazzinamento preferite e non viene effettuato nessun tentativo di conversione da una classe di immagazzinamento ad un'altra.

3.3 Progetto di una base di dati per la gestione dei clienti e delle lavorazioni in una piccola azienda

Lo scopo di questa base di dati è rappresentare i dati dei clienti, i preventivi da loro richiesti, i contratti di lavoro accettati dalla ditta e i pagamenti che avvengono a fine lavorazione, con relativa fatturazione.

3.3.1 Schema Entità-Relazione e schema logico

Si ha a disposizione il seguente schema E/R, già ristrutturato:

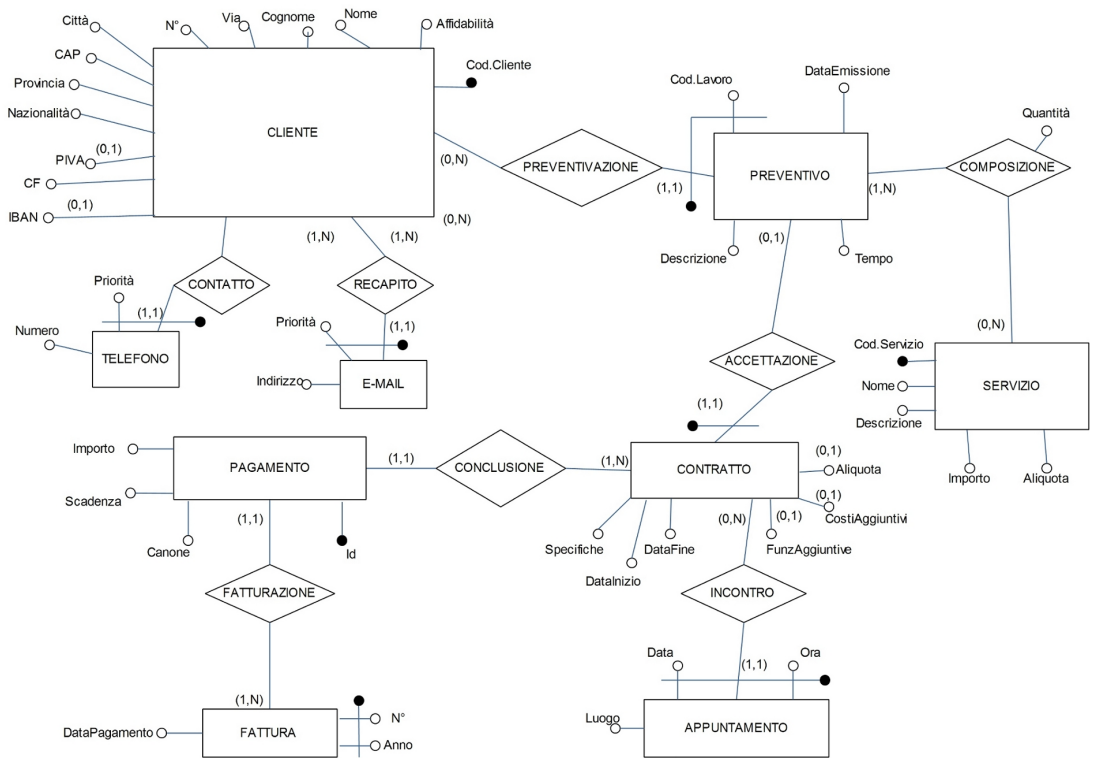


Figura 3.1: Schema E/R del database gestionale per piccole aziende

Da cui si ricava il seguente schema logico:

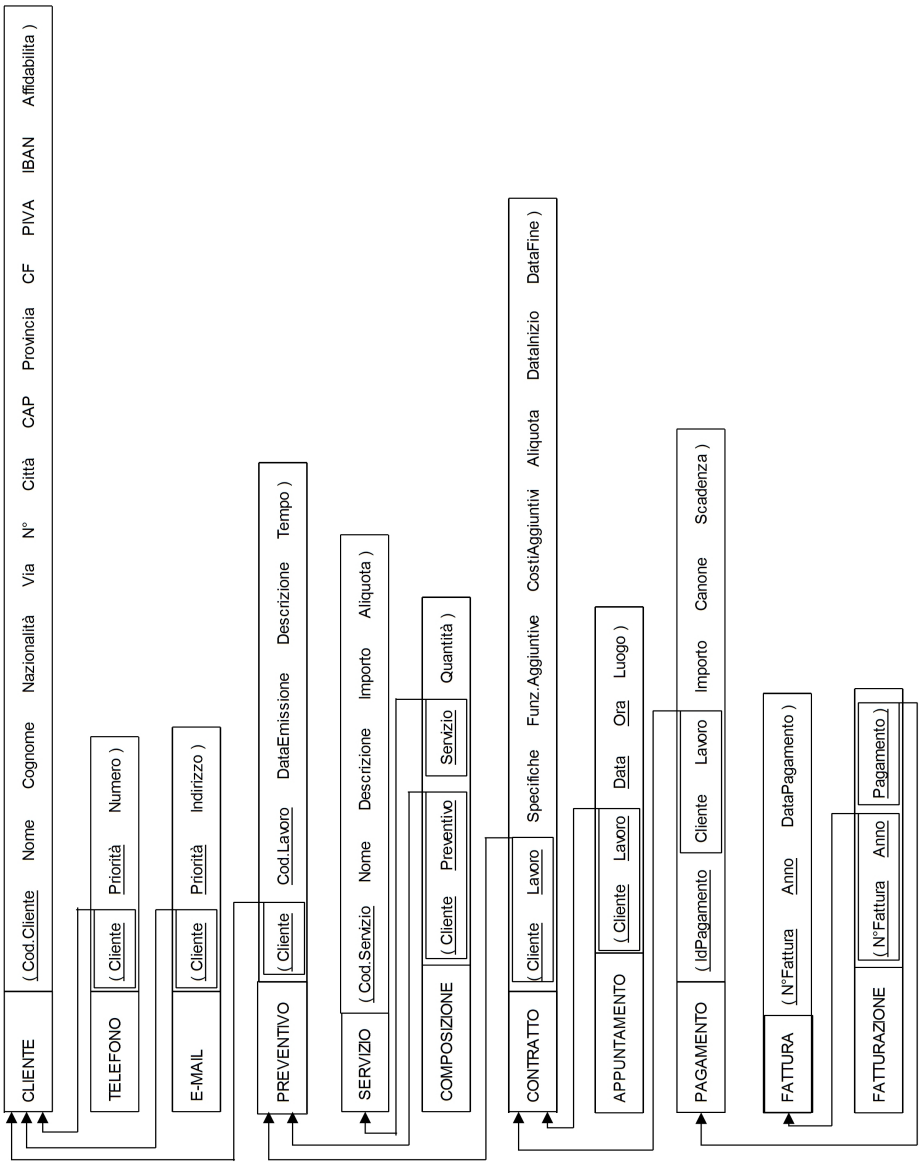


Figura 3.2: Schema Logico del database gestionale per piccole aziende

3.3.2 Codice SQL

Si aveva inoltre a disposizione il codice SQL per implementare la base di dati, opportunamente modificato per andare incontro alle esigenze di SQLite, quale ad esempio il tipo di dato *boolean* che ammette solo valori 0 e 1 invece dei canonici *true* e *false*.
Di seguito si riporta una porzione del codice SQL, dove sono presenti più componenti caratteristiche della scrittura in SQLite.

Codice 3.1: Porzione del codice SQL per la creazione delle tabelle

```
[...]
CREATE TABLE contratto
(
    cliente  INTEGER CHECK (cliente>0),
    lavoro INTEGER CHECK (lavoro>0),
    specifiche TEXT,
    funz_aggiuntive TEXT,
    costi_aggiuntivi DECIMAL(8,2) CHECK (costi_aggiuntivi>=0),
    aliquota INTEGER CHECK (aliquota>=0 AND aliquota<=100),
    data_inizio DATE,
    data_fine DATE,
    PRIMARY KEY(cliente,lavoro),
    FOREIGN KEY(cliente, lavoro)
          REFERENCES preventivo(cliente, n_preventivo)
          ON DELETE CASCADE ON UPDATE CASCADE,
    CHECK ((funz_aggiuntive IS NOT NULL
       AND costi_aggiuntivi IS NOT NULL AND aliquota IS NOT NULL)
      OR (funz_aggiuntive IS NULL AND costi_aggiuntivi IS NULL
       AND aliquota IS NULL)),
    CHECK (data_inizio < data_fine OR data_fine IS NULL)
);

CREATE TABLE appuntamento
(
    lavoro INTEGER CHECK (lavoro>0),
    cliente INTEGER CHECK (cliente>0),
    data DATE,
    ora TIME,
    luogo VARCHAR(100),
    PRIMARY KEY(lavoro, cliente, data, ora),
    FOREIGN KEY(lavoro, cliente)
              REFERENCES contratto(lavoro, cliente)
              ON DELETE CASCADE ON UPDATE CASCADE
);
```

```
CREATE TABLE pagamento
(
    _id INTEGER PRIMARY KEY AUTOINCREMENT,
    cliente INTEGER NOT NULL CHECK (cliente>0),
    lavoro INTEGER NOT NULL CHECK (lavoro>0),
    importo DECIMAL(8,2) NOT NULL CHECK (importo>0),
    canone BOOLEAN DEFAULT 0,
    scadenza DATE,
    FOREIGN KEY(cliente, lavoro)
                REFERENCES contratto(cliente, lavoro)
                    ON DELETE CASCADE ON UPDATE CASCADE,
    CHECK (((scadenza NOTNULL) AND (canone = 1))
       OR (canone = 0 AND scadenza IS NULL))
);
[...]
```

3.3.3 Creazione applicazione

Per prima cosa sono state create due interfacce contenenti stringhe costanti (definite come *public final static String*), una (**Preferences.java**) contenente il codice SQL mostrato in precedenza per la definizione del database e una (**Constants.java**) contenente il nome delle relazioni e dei relativi attributi, utilizzate in seguito per gli inserimenti e per un eventuale uso nelle interfacce grafiche.
Successivamente si è proceduto con la creazione della classe che gestisce la definizione e la manipolazione del database, **GestData.java**, che estende la classe **SQLiteOpenHelper** di Android, che contiene costruttori, metodi e variabili per la gestione di basi di dati; per fare tutto ciò però è necessario importare le classi **android.database.sqlite.SQLiteDatabase** e **android.database.sqlite.SQLiteOpenHelper**.
In questa classe viene definito il nome del database (tramite una stringa costante) e la versione del database, utile in caso di aggiornamenti futuri alla struttura che sovrascriveranno automaticamente il database precedente.
La definizione del database avviene al momento della creazione dell'istanza della classe, avviata automaticamente da Android con il metodo **onCreate**, attraverso il metodo **execSQL** della classe **SQLiteDatabase**, passando come parametro la stringa dove viene definita una tabella.

Codice 3.2: Esempio di definizione tabella

```
[...]
public void onCreate(SQLiteDatabase db) {

        db.execSQL(createTable_Cliente);
        Log.d("GESTDATA", "tabella clienti creata");
[...]
```

Come si può notare, dopo l'esecuzione dell'SQL si è chiamato un metodo della classe **Log**, che serve a visualizzare sul LogCat la coppia chiave/stringa che in questo caso viene utilizzata per notificare l'avvenuta creazione della tabella, utile in debug per non dover esaminare le molte eccezioni in cascata che si creano nel caso di codice Java o SQL errati, dovuti al sistema operativo stesso che propaga l'errore fino alla prima classe dalla quale derivano le altre.
Nel LogCat quindi vengono visualizzate le seguenti righe:

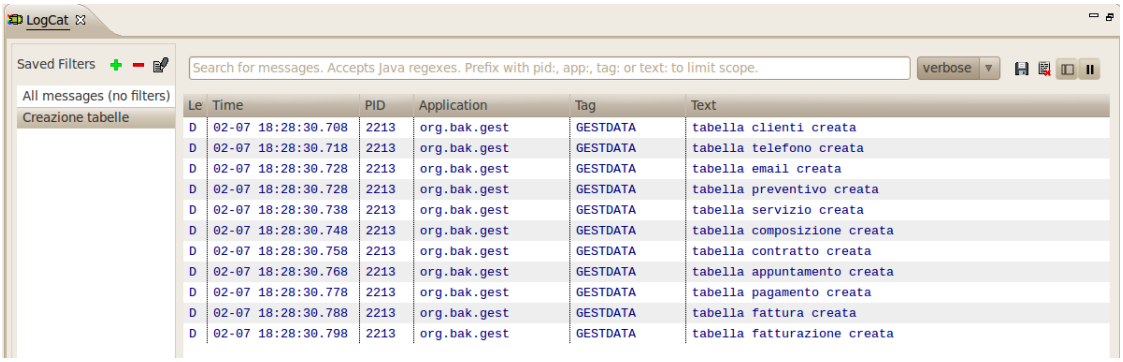


Figura 3.3: Visualizzazione su LogCat dell'avvenuta creazione delle tabelle

Per avviare l'applicazione, è necessario implementare la classe che coordina il tutto, **GestionaleActivity.java**, che si occupa di creare il database e di popolarlo, oltre a rendere disponibile un'interfaccia grafica per la visione delle tuple in una lista.
La creazione del database avviene, sempre all'interno del metodo onCreate, semplicemente creando un'istanza della classe GestData, vista in precedenza, che dev'essere obbligatoriamente chiusa. La popolazione del database, invece, viene effettuata all'interno di un costrutto *try...catch...finally* per evitare che l'insorgere di eccezioni possa forzare l'uscita dall'applicazione.

Codice 3.3: Metodo onCreate per la creazione e popolazione del database

```
[...]
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        gest = new GestData(this);
        try {
         populate();
        } finally {
         gest.close();
        }
    }
[...]
```

Nel codice si può notare che la popolazione viene effettuata con il metodo **populate()**, che non fa altro che chiamare altri metodi, come ad esempio **addAppuntamento**, che popolano ad una ad una le tabelle.

Codice 3.4: Metodo addAppuntamento per la creazione e popolazione del database

```
[...]
private void addAppuntamento(int cliente, int lavoro,
                    String data, String ora, String luogo) {
        SQLiteDatabase db = gest.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(CLIENTE_APP, cliente);
        values.put(LAVORO_APP, lavoro);
        if(data==null) values.putNull(DATA_APP);
        else {
            Date dat = Date.valueOf(data);
            SimpleDateFormat df =
                    new SimpleDateFormat("yyyy-MM-dd");
            values.put(DATA_APP, df.format(dat));
        }
        if(ora==null) values.putNull(ORA);
        else {
            Time time = Time.valueOf(ora);
            SimpleDateFormat tf =
                    new SimpleDateFormat("HH:mm:ss");
            values.put(ORA, tf.format(time));
        }
        if(luogo==null) values.putNull(LUOGO);
        else values.put(LUOGO, luogo);
        db.insertOrThrow(T_APPUNTAMENTO, null, values);
    }
[...]
```

Come si può notare, per prima cosa viene richiesto al gestore del database una copia modificabile dello stesso, attraverso il metodo **getWritableDatabase()**. Successivamente si crea un contenitore di valori che sarà poi utilizzato per effettuare direttamente l'inserimento della tupla, senza utilizzare codice SQL, ma attraverso il metodo del gestore **insertOrThrow(String table, String nullColumnHack, ContentValues values)** , il quale necessita di una tabella e di una serie di valori da inserire. Il parametro *nullColumnHack* è opzionale e serve ad inserire una riga vuota (operazione non permessa in SQL), inserendo come stringa il nome della colonna il cui valore può essere *null*.
In questo esempio inoltre si incontra un limite di SQLite, la gestione dei dati temporali. Per ovviare a questo problema, si è dapprima convertito il valore, passato come stringa, in un tipo di dato consono, come **Date** o **Time**, per poi eventualmente riformattarlo in maniera che le operazioni sul database possano

essere effettuate con la classe **SimpleDateFormat**, il cui costruttore accetta un *pattern*, ed il metodo **format**, che permette la riformattazione.
Gli inserimenti verranno sempre effettuati omettendo l'eventuale attributo del campo *_id*, in quanto autoincrementante (come definito nel codice SQL).
Terminata la popolazione di ogni tabella, viene effettuata un altra scrittura sul LogCat per rendere chiaro che l'operazione è avvenuta con successo, visualizzando come in precedenza le seguenti righe:

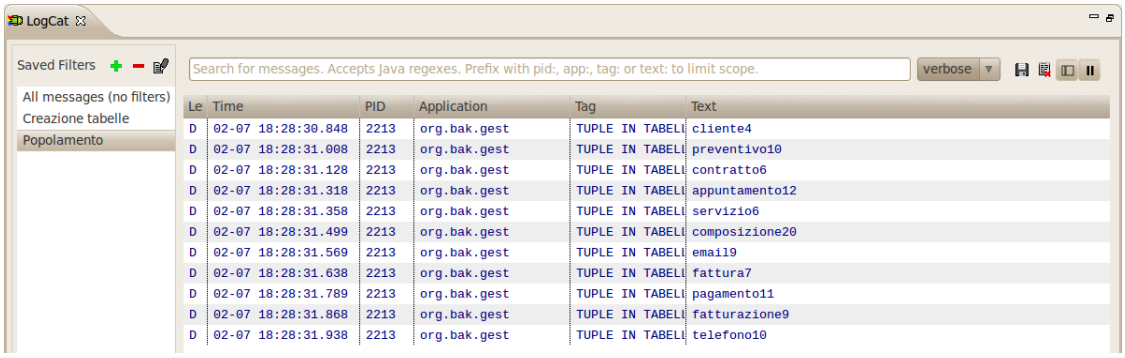


Figura 3.4: Visualizzazione su LogCat dei popolamenti

Al termine degli inserimenti si ha sul dispositivo questa visualizzazione delle tuple in una *ListView*.

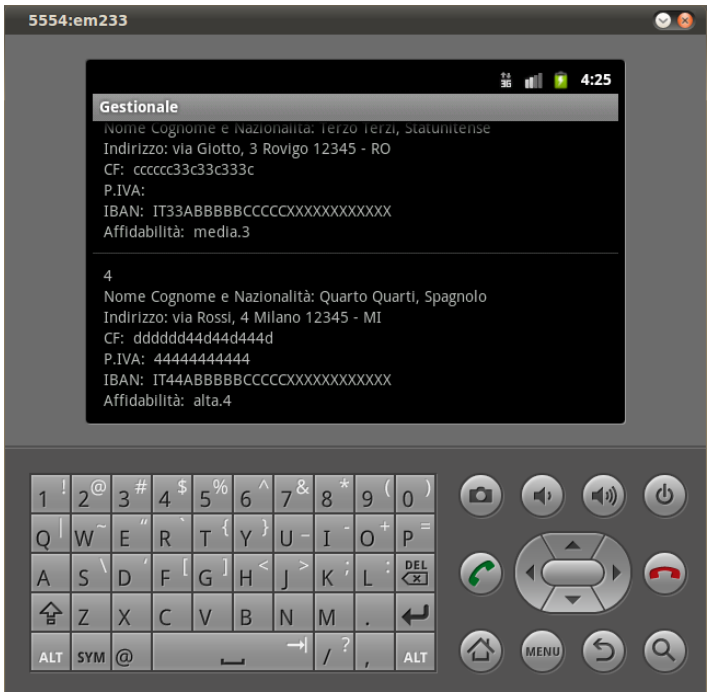


Figura 3.5: Visualizzazione sul dispositivo virtuale delle tuple nella relazione Cliente

3.4 Progetto di una base di dati per la gestione di esperimenti di information retrieval

In questo capitolo si tratterà della creazione di un database ampio e complesso per la gestione di esperimenti di *information retrieval*, incentrato sulla catalogazione di documenti.

3.4.1 Schema E/R

Si ha a disposizione il seguente schema E/R

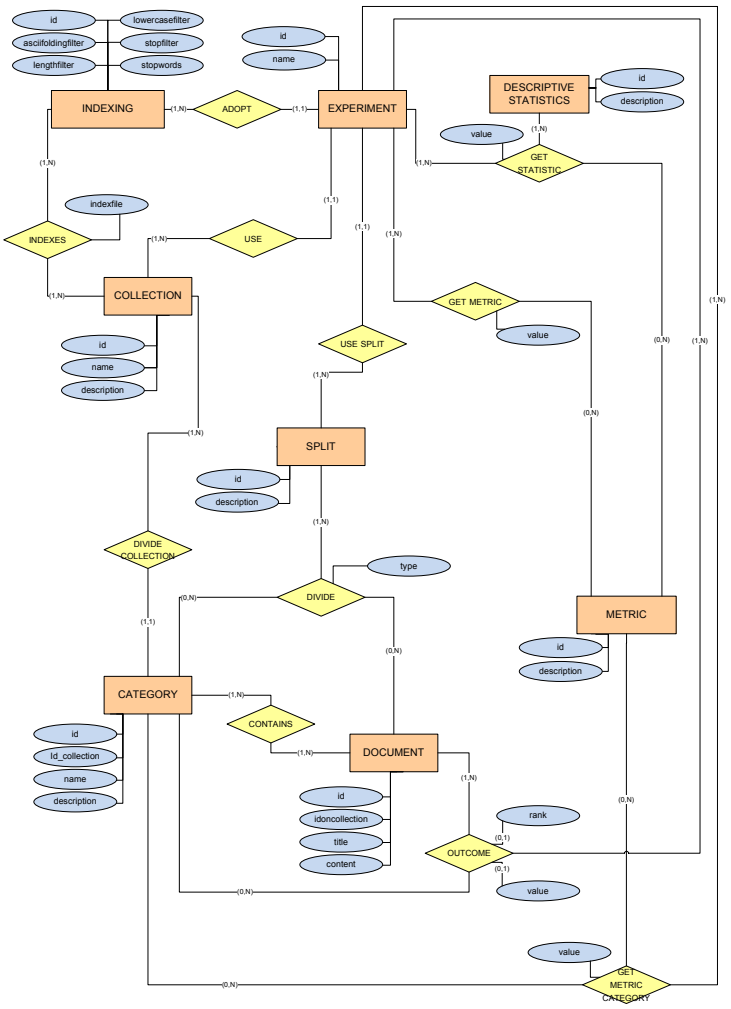


Figura 3.6: Schema E/R del database per information retrieval

3.4.2 Codice SQL

Si ha disponibile anche il codice SQL, del quale però si è utilizzato solo una minima parte per verificare la robustezza del sistema.

Codice 3.5: Porzione del codice SQL per la creazione delle tabelle

```
[...]
CREATE TABLE GE.Document
(
  id bigserial NOT NULL,
  idOnCollection character varying,
  title character varying,
  content text,
  CONSTRAINT PK_Documento PRIMARY KEY (id)
);
CREATE TABLE GE.Collection
(
  id serial NOT NULL,
  name character varying NOT NULL,
  description character varying,
  CONSTRAINT PK_Collection PRIMARY KEY (id)
);
CREATE TABLE GE.Category
(
  id serial NOT NULL,
  name character varying NOT NULL,
  id_collection serial NOT NULL,
  description character varying,
  CONSTRAINT PK_Category PRIMARY KEY (id ),
  CONSTRAINT FK_CategoryCollection FOREIGN KEY (id_collection)
      REFERENCES GE.Collection (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT
);
CREATE TABLE GE.Contains
(
  id_category serial NOT NULL,
  id_document bigserial NOT NULL,
  CONSTRAINT PK_Contains PRIMARY KEY (id_document , id_category),
  CONSTRAINT FK_ContainsCategory FOREIGN KEY (id_category)
      REFERENCES GE.Category (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT,
  CONSTRAINT FK_ContainsDocumento FOREIGN KEY (id_document)
      REFERENCES GE.Document (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT
);
[...]
```

La prima difficoltà incontrata in questa occasione è il fatto che SQLite non permette la creazione di schemi dentro i database, in quanto lo schema è concepito come l'insieme delle tabelle che compongono il database. Pertanto è stato necessario adattare il codice SQL fornito in modo da non dover creare uno schema.

3.4.3 Creazione applicazione

In questa applicazione non verranno implementate interfacce grafiche, essendo solo uno studio delle capacità del sistema, ma verrà analizzato il funzionamento con l'utilizzo del LogCat e della shell del sistema operativo sul dispositivo virtuale.
Per prima cosa si sono scritte tutte le righe di SQL in una serie di stringhe statiche costanti all'interno dell'interfaccia **Preferences.java**. Successivamente si è proceduto all'implementazione della classe che gestisce la creazione del database, **GestExpData.java**, che estende la classe **SQLiteOpenHelper** di Android.
Oltre alle classi **android.database.sqlite.SQLiteDatabase** e **android.database.sqlite.SQLiteOpenHelper**, questa volta è stato necessario importare anche la classe **android.os.Environment**, che si rende necessaria per creare il database nella *SDcard* anzichè nella memoria interna del dispositivo, per motivi di ingombro.

Codice 3.6: Classe GestExpData

```
[...]
public class GestExpData extends SQLiteOpenHelper{
   private static final String DATABASE_NAME=
                              "GestExperiment.db";
   private static final int DATABASE_VERSION=1;
   private static final String DB_PATH =
       Environment.getExternalStorageDirectory()+
             "/gestexp/"+DATABASE_NAME;

   public GestExpData(Context ctx) {
      super(ctx, DB_PATH, null, DATABASE_VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      db.execSQL(createTable_Document);
      Log.d("GESTEXPDATA", "tabella Document creata");
      db.execSQL(createTable_Collection);
      Log.d("GESTEXPDATA", "tabella Collection creata");
      db.execSQL(createTable_Category);
      Log.d("GESTEXPDATA", "tabella Category creata");
      db.execSQL(createTable_Contains);
[...]
```

Si può notare l'utilizzo che viene fatto della classe Environment del sistema operativo, per ottenere in maniera trasparente alla versione di Android il percorso dove è montata la SDcard, attraverso il metodo **getExternalStorageDirectory()**. Il percorso viene poi passato al costruttore della classe SQLiteOpenHelper che si occupa di creare il database nella posizione desiderata.
Successivamente si è implementata la classe **GestExperimentActivity.java**, nella quale ci si è occupati della popolazione del database.
La creazione delle tabelle avviene solo al momento della popolazione del database, per ovvie scelte di ottimizzazione fatte dal sistema operativo.
Al momento della creazione avvenuta di ogni tabella, come scritto nel codice, il LogCat visualizza i messaggi di controllo impostati.

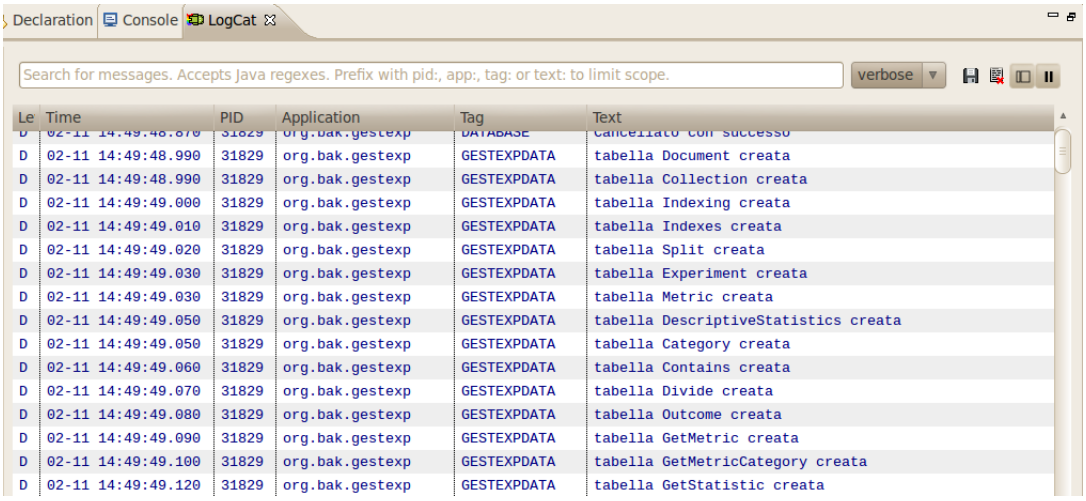


Figura 3.7: Visualizzazione su LogCat dei messaggi di creazione delle tabelle

Dovendo fare esperimenti sulle capacità del sistema, per prima cosa si sono creati dei metodi per ottenere stringhe casuali di varia lunghezza da utilizzare poi negli inserimenti nel database.

Codice 3.7: Metodi per ottenere stringhe casuali

```
[...]
private Random rnd = new Random();
private int TITLE_LENGTH=20;
private int TEXT_LENGTH=2000;
private int IDoC_LENGTH=10;
[...]
   private String randomTitle(){
      String titolo="titolo";
      for(int i=0; i<TITLE_LENGTH; i++){
         int offset=Math.abs(rnd.nextInt()%58);
         char letter = (char)('A' + offset);
         titolo+=letter;
      }
      return titolo;
   }
   private String randomContent(){
      String testo="testo";
      for(int i=0; i<TEXT_LENGTH; i++){
         int offset=Math.abs(rnd.nextInt()%58);
         char letter = (char)('A' + offset);
         testo+=letter;
      }
      return testo;
   }


   private String randomIdOnCollection(){
      String idOnCollection="id";
      for(int i=0; i<IDoC_LENGTH; i++){
         int offset=Math.abs(rnd.nextInt()%58);
         char letter = (char)('A' + offset);
         idOnCollection+=letter;
      }
      return idOnCollection;
   }
[...]
```

Nei metodi viene dapprima creata una stringa, per identificarne lo scopo, alla quale vengono poi concatenati tanti caratteri quanti la lunghezza desiderata, stabilita in partenza.

La casualità dei caratteri deriva dall'utilizzo di interi generati dalla classe **Random** di Java, i quali vengono ristretti ad un intervallo da 0 a 57 per poter utilizzare solo i caratteri *Unicode UTF-8* contenenti lettere latine maiuscole e minuscole e qualche simbolo. Il numero ottenuto viene poi sommato al carattere 'A' per restringersi all'intervallo di caratteri scelto.
Avendo a disposizione tali stringhe, si può ora procedere al popolamento parziale del database, attraverso i metodi **addDocuments()**, **addCollection()**, **addCategories()** e **linkDocsToCats()**.

Codice 3.8: Metodi per il popolamento

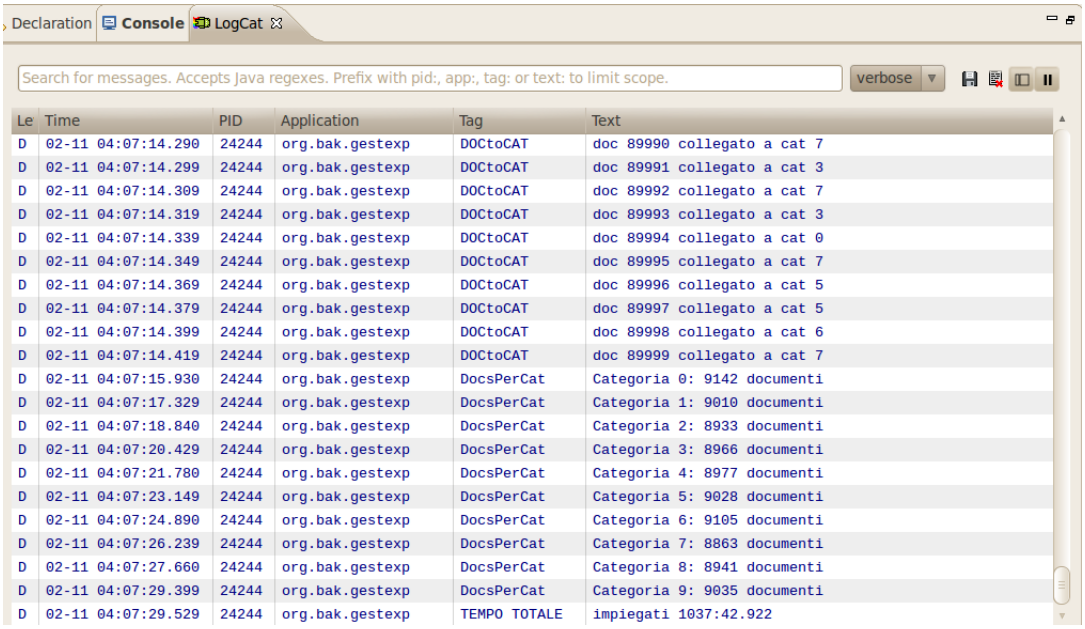
```
[...]
private int DOCS_NUMBER=90000;
private int CATS_NUMBER=10;
[...]
private void addDocuments() {
    for(int i=0; i<DOCS_NUMBER; i++){
        SQLiteDatabase db = gexp.getWritableDatabase();
        db.execSQL("INSERT INTO Document(id, idOnCOllection, "
            +"title, content) VALUES ("+i+", '"
            +randomIdOnCollection()+"', '"+randomTitle()+"', '"
            +randomContent()+"');");
        Log.d("DOCUMENTO", "inserimento "+i+" riuscito");
    }
}
private void addCollection() {
    SQLiteDatabase db = gexp.getWritableDatabase();
    db.execSQL("INSERT INTO Collection(id, name, description) "
        +"VALUES (0, '"+randomTitle()+"', '"+randomContent()+"');");
    Log.d("COLLEZIONE", "creazione riuscita");
}


private void addCategories(){
    SQLiteDatabase db = gexp.getWritableDatabase();
    for(int i=0; i<CATS_NUMBER; i++){
        db.execSQL("INSERT INTO Category(id, name, id_collection, "
            +"description) VALUES ("+i+", '"+randomTitle()+"', 0, '"
            +randomContent()+"');");
        Log.d("CATEGORIA", "creazione categoria "+i+" riuscita");
    }
}
```

```
private void linkDocsToCats(){
    SQLiteDatabase db = gexp.getWritableDatabase();
     for(int i=0; i<DOCS_NUMBER; i++){
        int cat = Math.abs(rnd.nextInt()%CATS_NUMBER);
        db.execSQL("INSERT INTO Contains(id_category, id_document) "
            +"VALUES ("+cat+","+i+");");
        Log.d("DOCtoCAT", "doc "+i+" collegato a cat "+c);
    }
}
[...]
```

In questa applicazione si è scelto di effettuare gli inserimenti lanciandoli direttamente come codice SQL per semplicità di scrittura.
Una volta effettuati gli inserimenti, è possibile visualizzare dalla *shell* del sistema operativo sul dispositivo virtuale i tipi di dato per cui è stata scelta l'affinità da SQLite.
In questo caso si trattava di interi per le colonne con attributo SERIAL o BIGSERIAL e di stringhe per le colonne con attributo CHARACTER VARYING. Questa operazione è possibile solo una volta effettuato il popolamento, in quanto la scelta del tipo di dato per affinità avviene al momento dell'inserimento, in base alle proprietà del dato.

Figura 3.8: Tipi di dato scelti per affinità

Alla fine delle operazioni si può quindi leggere da LogCat la quantità di documenti inseriti per categoria, distribuiti più o meno uniformemente tra le stesse.



Le	Time	PID	Application	Tag	Text
D	02-11 04:07:14.290	24244	org.bak.gestexp	DOCtoCAT	doc 89990 collegato a cat 7
D	02-11 04:07:14.299	24244	org.bak.gestexp	DOCtoCAT	doc 89991 collegato a cat 3
D	02-11 04:07:14.309	24244	org.bak.gestexp	DOCtoCAT	doc 89992 collegato a cat 7
D	02-11 04:07:14.319	24244	org.bak.gestexp	DOCtoCAT	doc 89993 collegato a cat 3
D	02-11 04:07:14.339	24244	org.bak.gestexp	DOCtoCAT	doc 89994 collegato a cat 0
D	02-11 04:07:14.349	24244	org.bak.gestexp	DOCtoCAT	doc 89995 collegato a cat 7
D	02-11 04:07:14.369	24244	org.bak.gestexp	DOCtoCAT	doc 89996 collegato a cat 5
D	02-11 04:07:14.379	24244	org.bak.gestexp	DOCtoCAT	doc 89997 collegato a cat 5
D	02-11 04:07:14.399	24244	org.bak.gestexp	DOCtoCAT	doc 89998 collegato a cat 6
D	02-11 04:07:14.419	24244	org.bak.gestexp	DOCtoCAT	doc 89999 collegato a cat 7
D	02-11 04:07:15.930	24244	org.bak.gestexp	DocsPerCat	Categoria 0: 9142 documenti
D	02-11 04:07:17.329	24244	org.bak.gestexp	DocsPerCat	Categoria 1: 9010 documenti
D	02-11 04:07:18.840	24244	org.bak.gestexp	DocsPerCat	Categoria 2: 8933 documenti
D	02-11 04:07:20.429	24244	org.bak.gestexp	DocsPerCat	Categoria 3: 8966 documenti
D	02-11 04:07:21.780	24244	org.bak.gestexp	DocsPerCat	Categoria 4: 8977 documenti
D	02-11 04:07:23.149	24244	org.bak.gestexp	DocsPerCat	Categoria 5: 9028 documenti
D	02-11 04:07:24.890	24244	org.bak.gestexp	DocsPerCat	Categoria 6: 9105 documenti
D	02-11 04:07:26.239	24244	org.bak.gestexp	DocsPerCat	Categoria 7: 8863 documenti
D	02-11 04:07:27.660	24244	org.bak.gestexp	DocsPerCat	Categoria 8: 8941 documenti
D	02-11 04:07:29.399	24244	org.bak.gestexp	DocsPerCat	Categoria 9: 9035 documenti
D	02-11 04:07:29.529	24244	org.bak.gestexp	TEMPO TOTALE	impiegati 1037:42.922

Figura 3.9: Popolazione delle categorie

Con le scelte effettuate è stato possibile creare, in un dispositivo virtuale con 512 MB di ram, un database di 90,000 documenti con testo di poco più di 2,000 caratteri, per una dimensione totale di più di 200 MB in un tempo lungo (più di 17 ore). Bisogna però considerare il fatto che era il sistema operativo a farsi carico di creare colta per volta le stringhe casuali basandosi sulla classe Random di Java (che si sa essere abbastanza onerosa dal punto di vista computazionale). La stessa prova è stata poi effettuata sulla macchina host dell'emulatore, che con gli stessi dati di partenza ha concluso le operazioni in quasi 28 minuti.
Per separare il tempo di elaborazione delle stringhe da quello necessario al DBMS, è stata fatta una prova con un database di 200 documenti da più di 500,000 caratteri creato sulla macchina host, che per ricontare i documenti ha impiegato circa 4.45 s, mentre sull'emulatore sono stati necessari circa 7.66 s.
Non è stato possibile fare esperimenti con database più grandi perché la selezione di tutti i documenti comporta la creazione di un duplicato della tabella Document, con un conseguente raddoppio delle dimensioni del database.

Capitolo 4

Conclusioni

In conclusione, il DBMS SQLite su piattaforma Android offre moltissime possibilità proprio per la sua semplicità, permettendo di immagazzinare qualsiasi dato senza restrizioni, o potendo utilizzare definizioni di database scritte per altri DBMS quasi senza necessità di modifiche, grazie al concetto delle affinità. Tale pregio però può facilmente comportare difficoltà al momento dell'esecuzione di interrogazioni, che possono risultare non ben definite dovendo demandare controlli sui tipi di dato, per inserimenti o per dati restituiti da query, alle applicazioni che utilizzano la base di dati.
Un grosso limite sulla velocità di elaborazione e sulla quantità di dati immagazzinabili è dato dall'hardware sul quale SQLite è fatto funzionare, che spesso non è ad altissime prestazioni essendo concepito per dispositivi embedded.
Per ovviare in parte a queste problematiche, si è dovuto realizzare la base di dati più complessa nella memoria di massa del dispositivo virtuale (o SDcard), in quanto non è stato possibile cambiare le specifiche dell'emulatore, tranne appunto la dimensione della scheda SD.
Un'altra difficoltà incontrata nel creare un database di grandi dimensioni è stata il frequente intervento del *garbage collector* durante la creazione di lunghe stringhe casuali, che ha rallentato enormemente l'inserimento dei dati. Si è tentato di variare quindi la memoria cache del dispositivo, senza però avere successo. Si può quindi affermare che SQLite per Android può essere utilizzato per applicazioni complesse, a patto di disporre di un sistema hardware capace.
Possibili sviluppi futuri possono essere l'installazione delle applicazioni su dispositivi reali, per poter effettuare test più efficaci, e la realizzazione di un'interfaccia grafica più completa.

Bibliografia

[1] The Pragmatic Programmers: Hello, Android - Introducing Google's Mobile Development Platform Third Edition, Ed Burnette, (2010)

[2] Android Market, `https://market.android.com`

[3] Guida all'installazione SDK Android, `http://developer.android.com/sdk/installing.html`

[4] Download Java Development Kit, `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

[5] Download Eclipse IDE, `http://www.eclipse.org/downloads/`

[6] Repository Android per Eclipse, `https://dl-ssl.google.com/android/eclipse/`

[7] Download Android Software Development Kit, `http://developer.android.com/sdk/index.html`

[8] Documentazione sui tipi di dato in SQLite, `http://www.sqlite.org/datatype3.html`