



Laboratorio

Android: gestione dei dati

Corso di Smart City e Tecnologie Mobili

Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

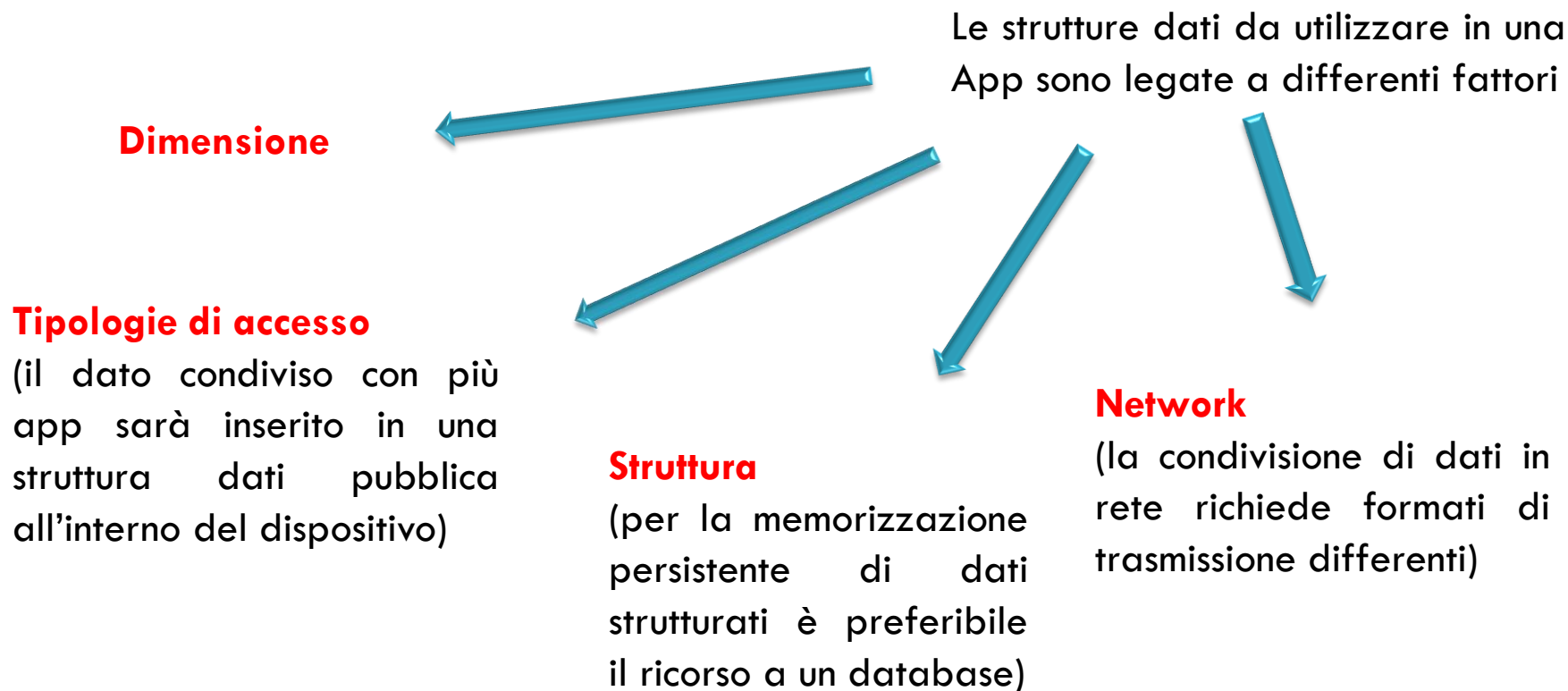
Dario Maio

Obiettivi



- Realizzazione di un'applicazione con gestione di database.
- Introduzione alle differenti tipologie di memorizzazione persistente del dato.
- SQLITE: introduzione ai database in Android.
- Architettura, implementazione e gestione delle strutture dati persistenti.
- Evoluzione del sistema: la gestione dei dati PRE e POST *HoneyComb*.
- SQLite Fast Insert con *Android Transactions*.

Necessità di una corretta gestione dei dati



Tipologie di memorizzazione dati

Android permette di utilizzare differenti tipologie di memorizzazione persistente dei dati:

1

SHARED PREFERENCES

2

INTERNAL STORAGE

3

EXTERNAL STORAGE

4

NETWORK STORAGE

5

DATABASES

Shared Preferences

La classe *SharedPreferences* mette a disposizione un framework per il salvataggio persistente e il rispettivo recupero di dati con le seguenti caratteristiche:



I dati sono in un formato di base (int, long, string, boolean, float).



I dati sono memorizzati in un formato dictionary *chiave-valore*.



Le Shared Preferences sono persistenti attraverso le sessioni utente: in caso di arresto improvviso dell'app il dato persiste.

Internal Storage e Cache

File dati possono essere salvati nello **Storage interno** del device oppure, in caso di file temporanei, nella **Cache** del sistema



I file restano accessibili solo dall'app di riferimento (in seguito a disinstallazione vengono cancellati).



Nel caso in cui il S.O. necessiti di spazio, può cancellare i file dell'app e svuotare la cache. Sono comunque presenti alcuni metodi per gestire queste tipologie di cancellazione.

External Storage

L'External Storage è un dispositivo di memorizzazione presente all'interno del device, ma accessibile dall'esterno. Non è necessariamente una SD Card removibile: i dispositivi a Storage fisso si stanno diffondendo rapidamente.



È accessibile dall'esterno: possono accedervi altre app (es: un lettore di File System) e può essere montato su un computer come dispositivo di memorizzazione di massa removibile.



Deve essere verificata la sua disponibilità: se è utilizzato da sistemi esterni potrebbe essere in uno stato non accessibile.

Network Connections

Sono una tipologia di Storage formato da una serie di servizi web finalizzati alla memorizzazione e condivisione delle informazioni.



Storage di file system con opzioni di **sharing** tra gli utenti (Google Drive, DropBox, ...).



Condivisione di foto, filmati, generalità (Facebook, Google + ...)



È incluso in questa tipologia di storage qualsiasi servizio di rete, anche ad hoc, che permetta disponibilità e fruibilità delle informazioni al di fuori dello smartphone tramite un protocollo di rete.

Database

È l'ultima tipologia di storage di dati, ma con struttura differente e con una maggiore evoluzione dei costrutti di gestione.



Permette di organizzare i dati in DB relazionali e non, anche con strutture complesse.



Occupa poche risorse.



È comunque una tipologia di database pensata per dispositivi portatili, in movimento, non sempre connessi e con spazi limitati; per cui va escluso il paragone i DMBS normalmente ospitati su server, quali Oracle, SQL Server , mySQL, ecc.

SQLite Database Engine

Con un utilizzo di risorse mediamente compreso fra i 4 e i 350k, è il motore di database transazionale Open Source più diffuso al mondo. Inoltre è:

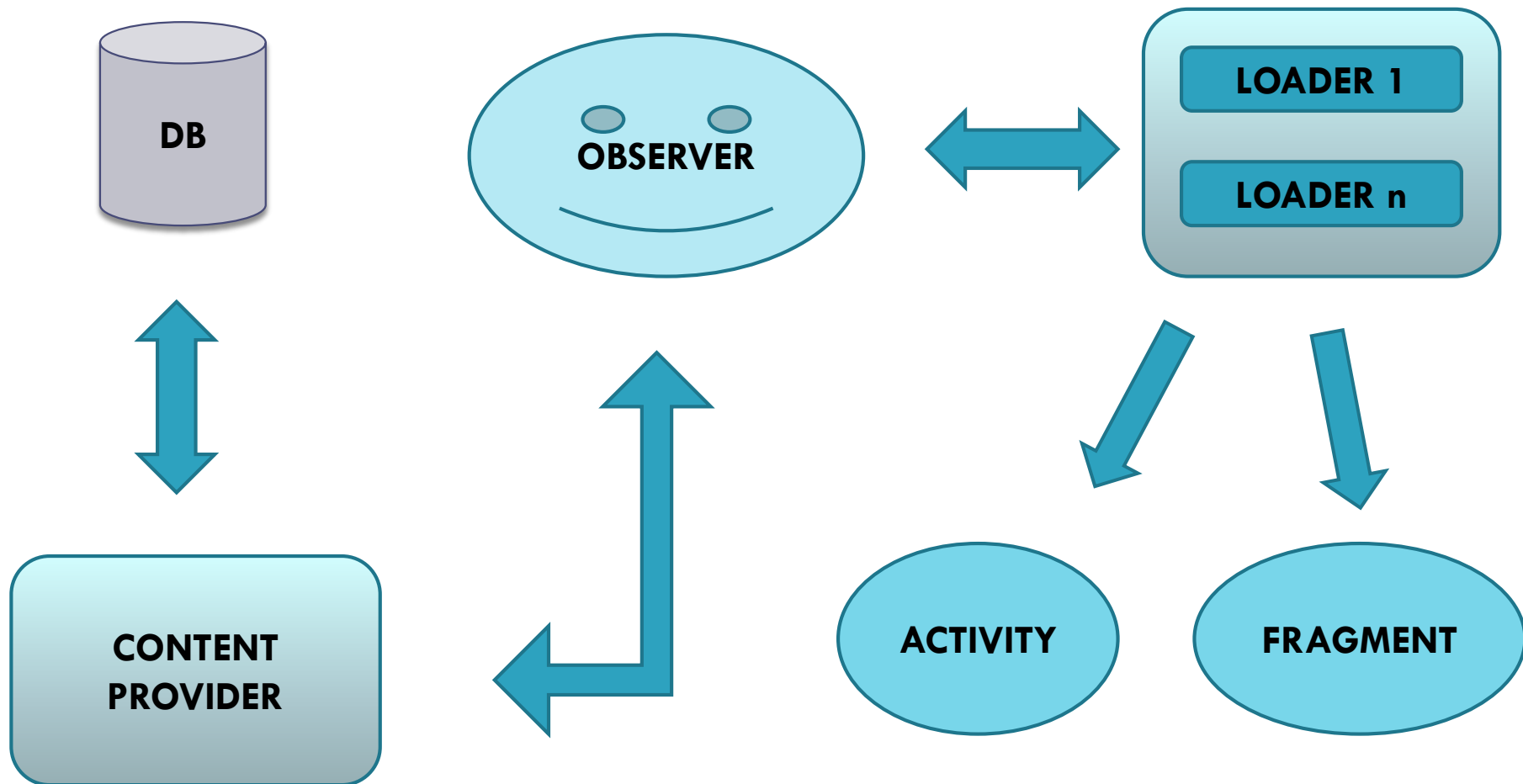
SELF CONTAINED: è scritto in ANSI C e limita al massimo il supporto a librerie esterne.

SERVERLESS: non ha un servizio che soddisfa le richieste, ma le applicazioni accedono direttamente al file del database.

ZERO CONFIGURATION: non esistono procedure di installazione, servizi da abilitare o file di configurazione.

TRANSACTIONAL: è un DBMS transazionale, quindi rispetta le proprietà ACID delle transazioni (Atomicità, consistenza, Isolamento, Durabilità).

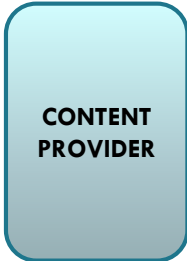
SQLite e Android – Architettura (1/2)



SQLite e Android – Architettura (2/2)



È definito da una classe (SQLiteOpenHelper) che implementa la struttura e i metodi di creazione delle tabelle.



Costituisce il primo livello di isolamento: definisce i metodi per accedere ai dati contenuti nel DB (SELECT, INSERT, DELETE e UPDATE). Permette di accedere a qualsiasi struttura (Mail, Musica, ecc ...): in questa particolare casistica si implementano peculiari metodi di accesso a un database.



Loader e Observer permettono il passaggio dati **ASINCRONO** tra il database e i fragment/activity. Il Loader si occupa del traferimento dati e Observer ha il compito di monitorarne eventuali variazioni e segnalarle al Loader per aggiornamenti successivi dei nostri fragment/activity.



Compongono la User Interface: le schermate con le liste, i pulsanti, gli spinner ecc.

Implementazione della Struttura Dati (1/2)

La struttura del DB viene creata implementando un'estensione della classe SQLiteOpenHelper

```
public class DBStructure extends SQLiteOpenHelper{

    private int mDBVersion=1;

    //Costruttore della classe
    public DBStructure(Context context, String name, CursorFactory factory,int version)
    {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Di seguito richiamo i metodi di creazione delle tabelle
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(DBStructure.class.getName(), "Upgrading database from version " +
            oldVersion + " to " + newVersion + ", which will destroy all old data");
        //Richiamo dei metodi di upgrade delle tabelle
        mDBVersion++;
    }
}
```

Successivamente nella classe si procederà alla creazione di alcuni metodi di creazione/cancellazione delle tabelle e di reset del DB

Implementazione della Struttura Dati (2/2)

La tabelle sono delle classi **static** che ne definiscono i campi e i comandi di base: **CREATE, DROP, INSERT, UPDATE, DELETE**, nel caso sottostante la tabella delle CURRENCY

```
public static class TCODCurrency{
    public static final String TB_NAME="T_COD_CURRENCY";
    public static final String IDCURRENCY = "_id";
    public static final String DESCURRENCY="des_currency"; //currency description (ex. Euro)
    public static final String TCODCURRENCY_TABLE_INSERT="INSERT INTO "+TCODCurrency.TB_NAME+" ("
        + TCODCurrency.IDCURRENCYSTATUS + ", "+ TCODCurrency.DESCURRENCY + ", "+
        TCODCurrency.DESCURRENCYSYMBOL + ", "+ TCODCurrency.TMS + ") "+ "VALUES (?,?,?,?)";
    public static final String TCODCURRENCY_TABLE_DELETE_ALL="DELETE FROM "+TCODCurrency.TB_NAME;
    // ...
}
//Table T_COD_CURRENCY creation query definition
private static final String TCODCURRENCY_TABLE_CREATE="CREATE TABLE IF NOT EXISTS "
    + TCODCurrency.TB_NAME + "("
    + TCODCurrency.IDCURRENCY + " integer primary key autoincrement, "
    + TCODCurrency.IDCURRENCYSTATUS + " integer not null, "
    + TCODCurrency.DESCURRENCY + " text not null,"
    + TCODCurrency.DESCURRENCYSYMBOL + " text not null,"
    + TCODCurrency.TMS+" text not null)";
//Table T_COD_CURRENCY drop table query definition
private static final String TCODCURRENCY_TABLE_DROP="DROP TABLE IF EXISTS " + TCODCurrency.TB_NAME;
```

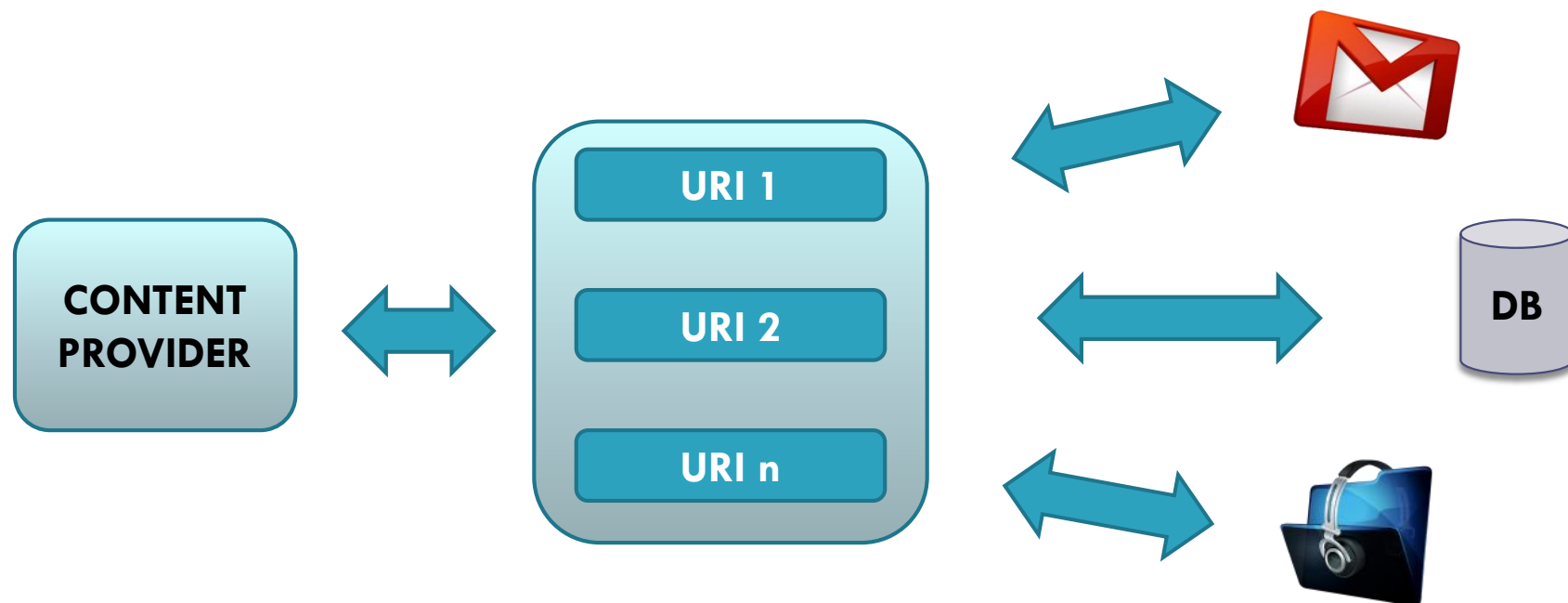
Le stringhe contenenti il comando SQL saranno in seguito eseguite tramite **db.execSQL(String)**

Ex. **db.execSQL(TCODCURRENCY_TABLE_DROP)** per l'eliminazione della tabella.

Il Content Provider (1 / 3)

Il Content Provider è un oggetto che gestisce l'accesso a una risorsa generica. Quest'ultima non deve essere obbligatoriamente un DB, ma può trattarsi di una qualsiasi struttura (ex. Casella Mail, elenco di file audio ...).

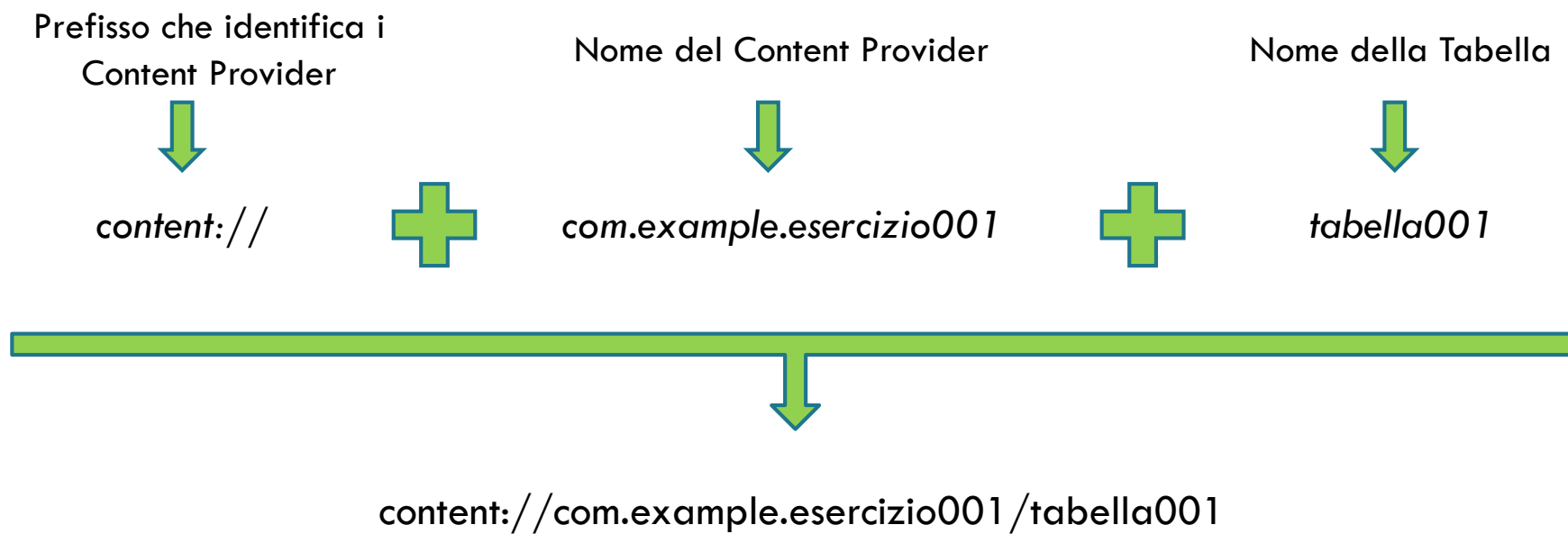
L'accesso alle risorse da parte del Content Provider avviene tramite un oggetto URI (**Universal Resource Identifier**)



Il Content Provider (2/3) – L'URI

Come citato da RFC 2396: “A *Uniform Resource Identifier (URI)* is a compact string of characters for identifying an abstract or physical resource”, cioè una stringa che identifica in modo univoco una risorsa del sistema.

Struttura dell'URI:



Il Content Provider (3/3) – Implementazione

Un semplice Content Provider di accesso a un DB avrà al suo interno l'implementazione dei 4 metodi fondamentali per accedere alle principali tabelle di riferimento in uso nell'app.

SELECT



```
switch
(UriMatcher.match(uri)){
case uri1: db.query(...)
    break;
case uri2: db.query(...)
    break;
```

INSERT



```
switch
(UriMatcher.match(uri)){
case uri1: db.insert(...)
    break;
case uri2: db.insert(...)
    break;
```

DELETE



```
switch
(UriMatcher.match(uri))
{
case uri1:
db.delete(...)
    break;
case uri2:
db.delete(...)
    break;
```

UPDATE



```
switch
(UriMatcher.match(uri)){
case uri1:
db.update(...)
    break;
case uri2:
db.update(...)
    break;
```

Prima di HoneyComb (1 / 2)

La maggiore evoluzione nel sistema di gestione dei dati in Android si è verificata col passaggio da **GingerBread** (2.3) a **HoneyComb** (3.0). Prima di HoneyComb la gestione dei dati era demandata ai seguenti metodi:


- *public void startManagingCursor(Cursor)*
- *public Cursor managedQuery(..)*

startManagingCursor(Cursor): lega la vita del cursore al ciclo di vita dell'activity chiamante. Il cursore è attivato, disattivato e chiuso a seconda che l'activity sia attiva, inattiva o sia distrutta. A ogni riattivazione dell'activity viene chiamato un metodo di *requery()* per aggiornare il cursore.

managedQuery(..): popola il cursore, collegato all'activity tramite il metodo precedente, con l'estrazione dati definita nei parametri.

Prima di HoneyComb (2/2)


Problematiche legate ai metodi precedenti:



Gestione delle query nel UI Thread con conseguente blocco dell'interfaccia utente nel caso di estrazioni di grandi volumi di dati.



Requery del cursore ogni qualvolta l'activity ritornava "viva" da uno stato di inattività.



Il SimpleCursorAdapter (oggi deprecato) generava un Observer sui dati sottostanti che causava un Requery a ogni cambiamento. Poiché tutta la gestione era demandata al UI Thread le prestazioni dell'app degradavano notevolmente.

Loader (1/2)

Con la rapida e crescente complessità delle architetture e delle app, la situazione precedente non era più sostenibile. La risposta a questa sempre maggiore necessità di ottimizzazione delle risorse e delle performance sono i **Loader**.

CARATTERISTICHE

Astrazione: il dato viene incapsulato nel Loader e i metodi di caricamento di quest'ultimo non sono di competenza del Fragment o dell'Activity che li richiede.

Asincronismo: l'Activity/Fragment che usa il Loader non deve preoccuparsi di chiamarlo in un Thread separato. Il Loader è asincrono per definizione.

Event-Driven: all'interno del Loader è presente un Observer che monitora il Data Layer sottostante. A ogni aggiornamento di quest'ultimo, i rispettivi dati all'interno di Activity/Fragment collegati vengono anch'essi aggiornati.

Loader (2/2)

COMPONENTI

Task asincrono: è il task che esegue il caricamento dati in modalità asincrona, separato da UI Thread. È implementato estendendo la classe *AsyncTaskLoader<D>*.

Listener: a ogni Loader è collegato un Listener al quale sono consegnati i dati al termine del caricamento.

Observer: all'interno di ogni Loader deve essere implementato un Observer che monitori il Data Layer sottostante in modo da permettere l'aggiornamento delle informazioni in caso di cambiamenti.

3 STATI: la vita di un Loader è scandita dai seguenti stati di attività:

1 – *STARTED STATE*: il Loader in questo stato esegue il caricamento dati, consegna i dati caricati al client chiamante e monitora i cambiamenti nel Data Layer.

2 – *STOPPED STATE*: il Loader in questo stato NON esegue il caricamento dati, NON effettua consegne di dati al chiamante, ma continua a monitorare i cambiamenti nel Data Layer.

3 – *RESET STATE*: il Loader in questo stato NON esegue il caricamento dati, NON effettua consegne di dati al chiamante e NON continua a monitorare i cambiamenti nel Data Layer. Inoltre rimuove qualsiasi dato caricato e non lo rende più disponibile.

Il Loader Manager (1 / 2)

I Loader sono degli ottimi esecutori dell'attività di estrazione e aggiornamento delle informazioni ma hanno bisogno di un coordinatore: **il LOADER MANAGER**

CARATTERISTICHE

È il responsabile della gestione dei Loader e ne esiste uno per ogni Activity/Fragment che li utilizza.

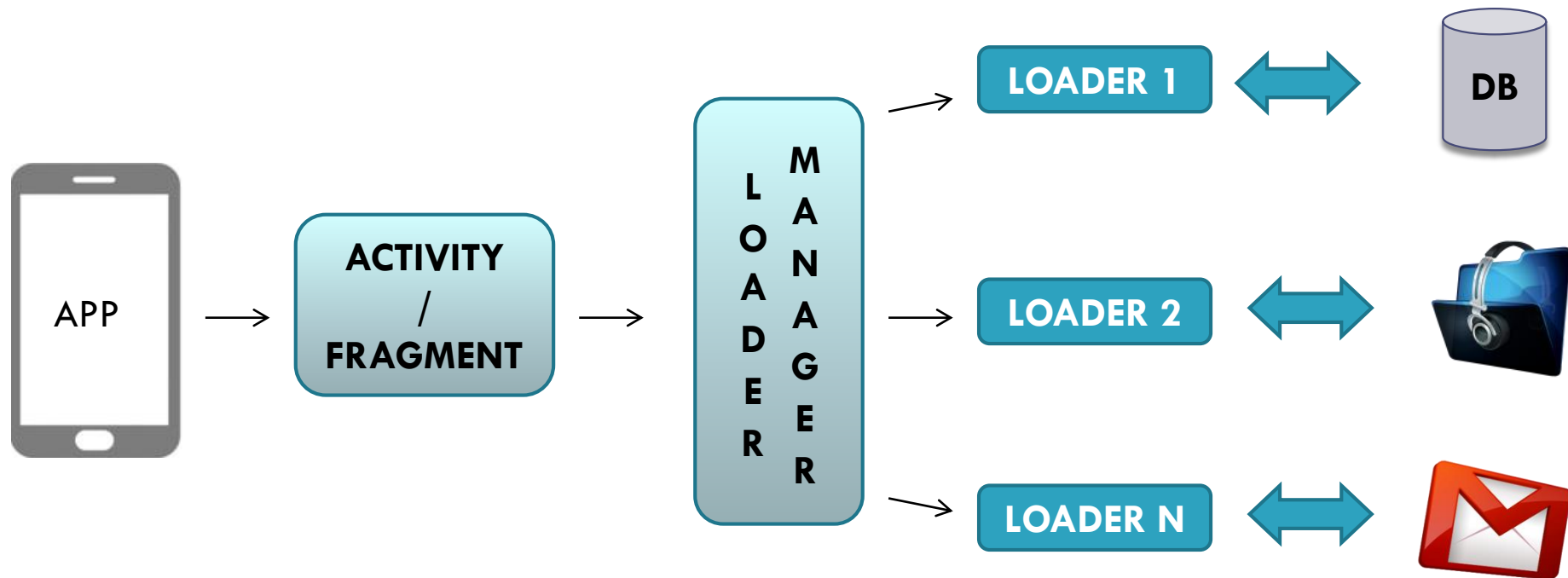
Effettua sui Loader le operazioni di:

- inizializzazione: *initLoader()*;
- riavvio: *restartLoader()*;
- eliminazione: *destroyLoader()*.

Effettua unicamente attività di coordinamento: non è competenza del Loader Manager la metodologia di implementazione dell'estrazione dei dati dei singoli Loader o del Data Layer sottostante.

Il Loader Manager (2/2)

Rappresentazione grafica dell'architettura con l'utilizzo dei Loader



SQLite Fast Insert: le Transazioni (1 / 3)

Normalmente gli inserimenti di dati in un Database SQLite sono di pochi record alla volta, ad esempio per un nuovo contatto, una nota, o il singolo elemento in questione dell'App che è in uso in un certo momento.

Se consideriamo, ad esempio, l'inserimento di un contatto nel Database DBContatti della nostra agenda, ipotizzando di avere implementato una tabella CONTATTI, avremo un'unica Transaction di INSERT come nell'esempio seguente:

```
ContentValues values = new ContentValues();  
values.put(CONTATTI.nome, "Paolo");  
values.put(CONTATTI.cognome, "Bianchi");  
values.put(CONTATTI.numero, "3389658741");  
DBContatti.insert(CONTATTI,null,values);
```



SQLite Fast Insert: le Transazioni (2/3)

... ma se i contatti da inserire sono più numerosi, il discorso si complica.

Consideriamo un `ArrayList` `contatti` di oggetti `Contatto{String nome, String cognome, String numero}` e inseriamoli in `dbContatti` utilizzando la sintassi precedente

```
for(int i=0;i<contatti.size();i++){  
    ContentValues values = new ContentValues();  
    values.put(CONTATTI.nome, contatti[i].nome);  
    values.put(CONTATTI.cognome, contatti[i].cognome);  
    values.put(CONTATTI.numero, contatti[i].numero);  
    dbContatti.insert(CONTATTI,null,values);  
}
```



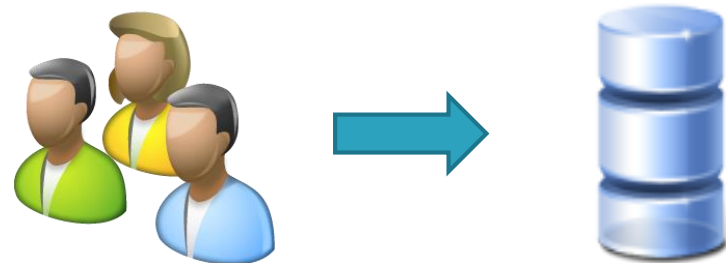
Sintatticamente corretto, ma richiede $N = \text{contatti.size}()$ Transazioni di INSERT e di conseguenza N accessi al Database e allo storage del dispositivo.

Per quanto le architetture di storage possano essere ottimizzate su uno smartphone si ha comunque un notevole decremento delle prestazioni.

SQLite Fast Insert: le Transazioni (3/3)

La soluzione: effettuare tutti gli inserimenti in un'unica transazione utilizzando la classe `SQLiteStatement`

```
dbContatti.beginTransaction();
String sqlInsert="INSERT INTO CONTATTI .."
SQLiteStatement sqlStmt =
dbContatti.compileStatement(sqlInsert);
for (int i = 0; i < contatti.size(); i++) {
    sqlStmt.bindString(1, contatti[i].nome);
    sqlStmt.bindString(2, contatti[i].cognome);
    sqlStmt.bindString(3, contatti[i].numero);
    sqlStmt.execute();
    sqlStmt.clearBindings();
}
dbContatti.setTransactionSuccessful();
dbContatti.endTransaction();
```



In questo modo si riduce il tutto a un'unica transazione incrementando le performance.

È fondamentale che questo tipo di operazioni siano implementate in un task asincrono al di fuori del UI Thread in modo da essere eseguite in background senza rallentare l'interfaccia utente.

Conclusioni

Android fornisce differenti tecnologie di memorizzazione dati differenziate in base all'utilizzo e alla tipologia del dato. Inoltre si deve tener ben presente che:



le risorse sono preziose e non devono essere sprecate con facilità;



è preferibile l'utilizzo di task asincroni per la gestione dei dati in modo che le operazioni siano eseguite in background;



le attività di gestione dei dati, soprattutto quelle che coinvolgono i database, devono essere effettuate al di fuori del UI Thread in modo da non sovraccaricare o addirittura bloccare l'interfaccia utente.



Le attività di inserimento che coinvolgono un elevato numero di record sono più efficienti se inserite in un'unica transazione.