SimpleDB, un DBMS didattico

17/04/2012

SimpleDB

- Sviluppato da Ed Sciore, professore al Boston College come strumento didattico per comprendere la struttura di un DBMS (e insegnarla)
- Scaricabile da http://www.cs.bc.edu/~sciore/simpledb/
- Versione base (in Java)
  - 3500 linee di codice
  - 85 classi
  - 12 package
- Estensioni circa 2000 altre linee di codice
- Nota: i sistemi reali open source (ad esempio Derby) sono cento volte più grandi; quelli commerciali ancora di più

SimpleDB, componenti

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Un'idea del funzionamento

- Un client JDBC sottomette una SELECT o DELETE o UPDATE scritta in SQL attraverso un metodo executeUpdate(…)
  - Remote riceve la chiamata e passa a Planner la stringa SQL
  - Planner invia la stringa a Parser che restituisce i nomi delle tabelle e le condizioni; con ciò, Planner può costruire un piano di esecuzione (espressione algebrica) che invia a Query
  - Query crea uno scan per ogni operazione nel piano e un Record Manager per ogni tabella coinvolta, che effettua la scansione (o altra forma di accesso)
  - Record gestisce i record conoscendone l'organizzazione in blocchi e quindi richiede a Transaction la lettura (o scrittura) di specifici dati
  - Transaction, se non ci sono problemi di concorrenza chiede a Buffer di leggere o scrivere
  - Buffer (coordinandosi con Log) stabilisce quali operazioni di lettura e scrittura servono davvero e le richiede a File
  - File legge e scrive (da disco verso pagina e viceversa)

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

File Manager

- Il componente di SimpleDB responsabile per l'interazione con il sistema operativo
- Una base di dati SimpleDB è memorizzata in vari file, per
  - tabelle
  - indici
  - log
  - catalogo
- Accesso a livello di blocco, dati il nome del file e il numero del blocco:
  - Si caricano, scrivono e appendono blocchi
  - Si leggono e modificano valori nelle pagine

File Manager, API

Block (identifica un blocco)

public **Block**(String filename, int blknum);
public String **fileName**();
public int **number**();

Page (mantiene il contenuto di un blocco, I/O buffer del s.o.)

public **Page**();
public int **getInt**(int offset);
public void **setInt**(int offset, int val);
… **getString**(...), **setString**(...),
public void **read**(Block blk);
public void **write**(Block blk);
public Block **append**(String filename);

FileMgr (gestisce l'interazione col s.o., con i nomi di directory e file)

public **FileMgr**(String dirname);
public boolean **isNew**();
public int **size**(String filename);

Block

- Identifica un blocco (ma non gestisce il contenuto, che è responsabilità di **Page**)
  - nome del file e numero del blocco
- Implementazione "naturale"
- Include anche metodi
  - equals, toString e hashCode

Page

- costanti BLOCK_SIZE (dimensione del blocco), INT_SIZE e metodo STR_SIZE(n) (rispettivamente, byte per un intero e per una stringa di lunghezza n)
- I metodi sono sincronizzati (cioè eseguibili da un solo thread alla volta)
- Contenuto memorizzato in un ByteBuffer (creato con un metodo factory allocateDirect … e con metodi per leggere e scrivere un array di byte in nella posizione corrente, che a sua volta può essere specificata con un metodo e che viene modificata dalle operazioni): accessi e modifiche a singoli dati operano su tale ByteBuffer (contents)
- Offre ai package superiori metodi per leggere, scrivere e inserire (append) blocchi in un file; implementa questi metodi con chiamate a metodi a livello di package di FileMgr

FileMgr

- Classe singolare, con istanza creata all'avvio
- Conosce la directory del db e mantiene una mappa dei file aperti (nome e file channel)
- Alla richiesta di un file se non è aperto lo apre
- Implementa (con metodi visibili solo nel package) i metodi di lettura, scrittura e inserimento di blocchi definiti in Page
- Esegue le effettive letture e scritture su file usando le classi File (rappresentazione astratta di un file), RandomAccessFile (in sostanza, il file) e FileChannel (uno strumento per accedere ad un file e trasferire dati in modo non solo elementare)

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Buffer Manager, API

BufferMgr (classe singolare, con istanza creata all'avvio)

public **BufferMgr**(int numbuffers);
public int **available**();
public void **flushAll**(int txnum);
public Buffer **pin**(Block blk);
public Buffer **pinNew**(String filename, PageFormatter fmtr);
public void un**pin**(Buffer buff);

Buffer (un elemento del buffer, corrisponde ad una pagina)

public int **getInt**(int offset);
public void int **setInt**(int offset, int val, int txnum, int lsn);
… **getString**(...), **setString**(...),
public Block **block**();

PageFormatter (interfaccia)

public void **format**(Page p);

Buffer

- Campi
  - contenuto (Page), blocco cui si riferisce, numero di pin (programmi che lo utilizzano), transazione che ha modificato e identificatore del record di log associato (vedremo)
- Metodi pubblici, abbastanza intuitivi (ribaltano le operazioni sulla pagina); le scritture debbono conoscere anche la transazione e il log (e aggiornarle)
- Metodi di package:

  void **flush**() (se modificato, viene scritto)

  void **pin**()

  boolean **isPinned**()

  boolean **isModifiedBy**(int txnum)

  void **assignToBlock**(Block b) (inizia con una flush())

  void **assignToNew**(String filename, PageFormatter fmtr)

PageFormatter

- È un'interfaccia:
  - i moduli sovrastanti debbono scegliere la sua implementazione, qui l'unica cosa rilevante è che arrivino pagine formattate

BasicBufferMgr

- Classe di supporto, con metodi tutti a livello di package o privati
- Implementa i metodi semplici (restituisce null se non trova un buffer utilizzabile)
  - void **flushAll**(txnum) scarica le pagine modificate da txnum
  - Buffer **pin**(blk) se trova un buffer che usa già blk lo restituisce altrimenti se trova una pagina libera la assegna al blocco altrimenti restituisce null
  - Buffer **pinNew**(filename,fmtr) simile, ma non cerca, crea
  - void **unPin**(buff)
  - private Buffer **findExistingBuffer**(blk) cerca il buffer per un certo blocco
  - private Buffer **chooseUnpinnedBuffer(**blk) cerca un buffer non utilizzato

Strategia di rimpiazzo (Buffer replacement strategy)

- Quale che sia la politica (steal o no-steal), si pone l'esigenza di scegliere la pagina libera da associare al blocco (l'ideale sarebbe mantenere nel buffer le pagine che, pur libere, potrebbero essere riutilizzate)
- Strategie:
  - naif (cerca la prima pagina libera)
  - FIFO (utilizza la pagina che è stata caricata da più tempo)
  - LRU (utilizza la pagina che è stata usata meno di recente)
  - clock (fa una scansione, come nel caso naif, ma non dall'inizio, bensì dalla pagina successiva a quella del rimpiazzo precedente)

Strategia di rimpiazzo (Buffer replacement strategy)

- SimpleDB usa
  - "no steal"
  - strategia naif

BufferMgr

- Delega a BasicBufferMgr i compiti semplici e gestisce il caso in cui non c'è un buffer utilizzabile
  - unpin(buffer) esegue anche una notifyAll() se libera un buffer
- I metodi **pin**(blk) e **pinNew**(filename,fmtr) sono più sofisticati e oltre a chiamare gli omonimi metodi di BasicBufferMgr pongono in attesa (con il metodo wait() della classe Object) il thread fino ad un eventuale notify() o notifyAll() sullo stesso oggetto (il BufferMgr è uno solo, quindi tutti i thread utilizzano lo stesso) oppure alla scadenza di un timeout

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Record Manager, classi

- Schema
  - gestisce lo schema di una relazione (nome degli attributi, tipo e lunghezza)
- TableInfo
  - gestisce le informazioni fisiche di una relazione: lo schema e la sua implementazione (il nome del file, la lunghezza dei record e l'offset di ciascun campo)
- RecordPage
  - gestisce i record di una pagina, eseguendo operazioni semplici su quello corrente
- RecordFormatter
  - implementa PageFormatter: inizializza una pagina vuota, preparata per record di un certo tipo (vedi TableInfo)
- RecordFile
  - Gestisce i record in un file …
- RID
  - Identificatore di un record in un file (blocco e id nel blocco)

Schema e TableInfo, API

Schema

```
public void addField(String fldname, int type, int length);
public void addIntField(String fldname);
public void addStringField(String fldname, int length);
public void add(String fldname, Schema sch);
public void addAll(Schema sch);
public Collection<String> fields();
public boolean hasField(String fldname);
public int type(String fldname);
public int length(String fldname);
```

TableInfo

```
public  TableInfo(String tblname, Schema schema);
public  TableInfo(String tblname, Schema schema, Map<String,Integer> offsets,
int recordlen);
public  String fileName();
public  Schema schema();
public  int offset(String fldname);
public  int recordLength();
```

Schema

- Gestisce lo schema di una relazione (nome degli attributi, tipo e lunghezza)
  - insieme di terne: nome del campo, tipo e lunghezza (a livello **logico**, in caratteri e non in byte)
  - utilizza una mappa con nome del campo come chiave e tipo e lunghezza come valore (classe privata FieldInfo); la lunghezza è posta a 0 per gli interi, ma il valore non viene mai utilizzato
- I metodi sono molti per semplicità, basterebbe un metodo add, anziché cinque
- Implementazioni dei metodi ovvie
- I tipi sono denotati da costanti intere, come definite in java.sql.Types; sono implementati INTEGER= 4 e VARCHAR=12

TableInfo

- Gestisce le informazioni **fisiche** di una relazione: lo schema e la sua implementazione (il nome del file, la lunghezza dei record e l'offset di ciascun campo)
- Due costruttori
  - per creare la struttura
  - per riusare una struttura già creata
- Gestisce:
  - nome, schema
  - offset dei campi (con una mappa) e lunghezza dei record (fissa)
- Implementazione dei metodi semplici (il primo costruttore un po' articolato: crea la mappa e scandisce i campi calcolando l'offset per ciascuno)

RecordPage, API

- Gestisce i record in una pagina; ci sono vari slot, ognuno dei quali ha un id ed è utilizzato o meno
- Ha un "record corrente:" next() si posiziona su uno utilizzato, insert() su uno libero, moveToId(id) su uno specifico

```
public RecordPage(Block blk, TableInfo ti, Transaction tx);
public void close();
public boolean next();
public int getInt(String fldname);
public String getString(String fldname);
public void setInt(String fldname, int val);
public void setString(String fldname, String val);
public void delete();
public boolean insert();
public void moveToId(int id);
public int currentId();
```

RecordPage, implementazione

- Gli slot per i record hanno lunghezza fissa, gli slot si trovano in base alla lunghezza e si usa un byte (basterebbe un bit) per indicare se ciascuno slot è occupato o meno
- Gli slot sono identificati da un numero d'ordine
- I metodi get e set sono non posizionali e "traducono" il nome in offset (usando TableInfo) che viene passato ai moduli sottostanti (Transaction che poi chiama Buffer che chiama Page) – vediamo il codice
- I metodi next() e insert() possono fallire (se non trovano alcun ulteriore slot valido, usato o libero, rispettivamente); utilizzano il metodo provato searchFor()

RecordFormatter

- implementa l'interfaccia PageFormatter:
  - void format(Page p)
    - inizializza una pagina vuota, preparata per record di un certo tipo, specificato da un TableInfo (argomento del costruttore)
    - marca come vuoti tutti i record e inizializza gli interi a 0 e le stringhe a vuote

RecordFile, API

Costruttore

public RecordFile(TableInfo ti, Transaction tx)

Metodi per cambiare la posizione corrente

public void beforeFirst()
public boolean next()
public void moveToRid(RID rid)
public void insert()
public void close()

Metodi che accedono al record corrente

public int getInt(String fldname)
public String getString(String fldname)
public void setInt(String fldname, int val)
public void setString(String fldname, String val)
public void delete()
public RID currentRid()

RecordFile

- Costruttore public RecordFile(TableInfo ti, Transaction tx)
  - Associa il file alla TableInfo e opera nell'ambito di una transazione
- get, set e delete operano sul record corrente e quindi chiamano semplicemente gli omonimi a livello di pagina
- next si muove nella pagina e, se questa finisce, si sposta alla successiva (con il metodo privato moveTo che chiude la pagina corrente); se non c'è pagina successiva restituisce false
- insert cerca di inserire nel blocco corrente, avanzando finché non trova posto oppure appendendo un blocco

Esercitazioni pratiche

- SimpleDB va sperimentato
  - dopo averlo installato (il che è semplice, visto va solo scompattato), utilizziamolo (vedere README.txt)
  - è un sistema client-server
  - avvio del server (su Windows, simile negli altri ambienti)
    - start rmiregistry
    - java simpledb.server.Startup studentdb

Esercitazioni pratiche, 2

- Client
  - simpleDB è accessibile via JDBC
  - il driver è nel package simpledb.remote e client e server possono essere su macchine diverse; per comodità lavoriamo su una sola
  - ci sono alcuni client distribuiti con il sistema (nella cartella studentClient) e possiamo usarli per provare il funzionamento
    - CreateStudentDB
    - StudentMajors
    - FindMajors (un argomento)
    - SQLInterpreter
    - ChangeMajor

Esercitazioni pratiche, 3

- Le classi client non fanno parte esplicitamente di package
  vanno eseguite dalla directory in cui si trovano

  > cd C:….\studentClient\simpledb

  > java CreateStudentDB

  – oppure con Eclipse
- Provare ad eseguire i vari programmi client, almeno
  – CreateStudentDB (per creare una base di dati e vedere che il tutto funzioni)
  – SQLinterpreter (per vedere l'SQL utilizzato e le sue limitazioni, brevemente descritte nel README.txt)

Esercitazioni pratiche, 4

- SimpleDB è fatto per essere modificato …
- Prima ancora testiamo le classi nei singoli package
- Allo scopo, due avvertenze:
  - spesso le classi di test debbono poter accedere a metodi a livello di package ("friend") e quindi è spesso necessario scrivere classi di test nei package
  - per testare i moduli di livello basso, non serve lanciare il server, ma è necessario chiamare alcuni metodi che costruiscono oggetti fondamentali del server (ad esempio il FileMgr o il BufferMgr)

Esercizi da svolgere, obbligatori

- SimpleDB utilizza la strategia naif per il rimpiazzamento dei buffer
- Implementare un'altra delle strategie (ad esempio la "clock" che è molto semplice) e mostrare che porta significativi benefici; notare anche che la versione fornita utilizza pochissimi buffer (8), per migliorare le prestazioni è necessario aumentarli, valutare quale può essere un numero adeguato per ottenere i benefici

Esercizi da svolgere, obbligatori (2)

- Utilizzando SimpleDB (o meglio, i suoi moduli di livello più basso), scrivere una classe di test che esegue alcune operazioni su un file.

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Metadata Manager

- Gestisce il catalogo, che contiene info su
  - Tabelle
  - Viste
  - Indici
  - Statistiche (queste solo in memoria)

Metadata Manager API

```
public MetadataMgr(boolean isnew, Transaction tx)
public void createTable(String tblname, Schema sch, Transaction tx)
public TableInfo getTableInfo(String tblname, Transaction tx)
public void createView(String viewname, String viewdef,
                                                    Transaction tx)
public String getViewDef(String viewname, Transaction tx
public void createIndex(String idxname, String tblname,
                                          String fldname, Transaction tx)
public Map<String,IndexInfo> getIndexInfo(String tblname,
                                                  Transaction tx)
public StatInfo getStatInfo(String tblname, TableInfo ti, Transaction tx)
```

Classi nel Metadata Manager

- MetadataMgr, singleton, coordina e delega (ad altre quattro classi singleton)
- TableMgr, gestisce tabelle:
  - creazione, salvataggio nel catalogo, lettura; gestisce anche le tabelle del catalogo stesso (creandole se il db è nuovo)
- IndexMgr, gestisce indici, analogamente
- ViewMgr, gestisce viste (memorizza la definizione testuale)
- StatMgr, calcola le informazioni all'avvio e le aggiorna periodicamente (nella versione esistente ogni 100 chiamate); alcune scelte sono naif

- IndexInfo (nome, tabella, campo, statistiche e transazione)
- StatInfo (numero di blocchi e numero di record e di valori diversi per ciascun campo [l'ultima non implementata])

Il catalogo

tblcat(TblName, RecLength)

fdlcat(TblName, FldName, Type, Length, Offset)

viewcat(ViewName, ViewDef)

idxcat(Indexname, Tablename, Fieldname)

Guardare sul sistema (con SQLInterpreter)

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Query Manager

- SimpleDB esegue le interrogazioni per mezzo di espressioni di una sorta di algebra relazionale
- Il Query Manager è in grado di eseguire queste operazioni; i package Planner e Parser di tradurre interrogazioni da SQL in algebra
- Un'interrogazione è gestita da uno "scan":
  - Scan è un'interfaccia
  - Ci sono implementazioni per ciascun operatore e per la scansione di tabelle:
    - ogni nodo dell'albero che rappresenta un'interrogazione è un oggetto scan

Interfaccia Scan

```
public interface Scan {
    public void beforeFirst();
    public boolean next();
    public void close();
    public Constant getVal(String fldname);
    public int getInt(String fldname);
    public String getString(String fldname);
    public boolean hasField(string fldname);
    }
```

- Simile a RecordFile, con alcune differenze:
  - non conosce lo schema, permette solo di verificare l'esistenza di campi
  - ha il metodo getVal() (che restituisce oggetti di un tipo interfaccia Constant, implementata da opportune classi wrapper)
  - non permette aggiornamenti (vedi UpdateScan), perché non tutti gli scan possono essere aggiornabili

Interfaccia UpdateScan

```
public interface UpdateScan extends Scan {
    public void setVal(String fldname, Constant val);
    public void setInt(String fldname, int val);
    public void setString(String fldname, String val);
    public void insert();
    public void delete();

    public RID getRid();
    public void moveToRid(RID rid);
    }
```

Classi che implementano Scan (un primo elenco)

- Mostriamo i costruttori:

  public SelectScan(Scan s, Predicate Pred);

  public ProjectScan(Scan s, Collection<String> fldList);

  public ProductScan(Scan s1, Scan s2);

  public TableScan(TableInfo ti, Transaction tx);
- Osservazioni:
  - Come detto, un oggetto per ogni nodo (operatori e tabelle)
  - I nodi operatore hanno argomenti a loro volta di tipo scan
  - SelectScan e TableScan implementano UpdateScan

TableScan

- Implementa UpdateScan
- Usata per le foglie dell'albero, ha un oggetto RecordFile, creato sulla base del TableInfo passato al suo costruttore
- L'esecuzione dei metodi è delegata all'oggetto RecordFile (con opportuna gestione per getVal() e setVal(), vediamo)

TableScan, qualche dettaglio

```
public class TableScan implements UpdateScan {
   private RecordFile rf;
   private Schema sch;
public TableScan(TableInfo ti, Transaction tx) { … }
public void beforeFirst() { rf.beforeFirst(); }
public boolean next() { return rf.next();}
public Constant getVal(String fldname) {
      if (sch.type(fldname) == INTEGER)
         return new IntConstant(rf.getInt(fldname));
      else
         return new StringConstant(rf.getString(fldname));
   }
…
```

SelectScan

- Operatore di selezione
- Anche questa classe implementa UpdateScan
- Delega quasi tutto allo scan "sottostante" (che può essere un UpdateScan)
- L'unico metodo non banale è il metodo **next**, che restituisce il primo record successivo (se c'è) che soddisfa il predicato (di Predicate parliamo più avanti), vediamo

SelectScan, qualche dettaglio

```
public class SelectScan implements UpdateScan {
   private Scan s;
   private Predicate pred;
public SelectScan(Scan s, Predicate pred) { … }
…
public boolean next() {
      while (s.next())
         if (pred.isSatisfied(s))
         return true;
      return false;
   }
…
```

ProjectScan

- Realizza l'operatore di Proiezione.
- La lista di campi è passata al costruttore ed è utilizzata dal metodo hasField.
- Gli altri metodi semplicemente inoltrano la richiesta allo scan sottostante.
- È l'unico scan che nella versione base effettua controllo di esistenza del campo per i metodi di get (non può semplicemente inoltrare allo scan sottostante)

ProductScan

- Realizza il prodotto cartesiano:
  - ha due scan interni e li combina con un nested-loop
  - metodo parte dal primo record di s1 (vedi costruttore) e itera attraverso s2, poi raggiunta la fine di s2, passa al secondo record di s1 e itera su tutto s2
  - Il metodo next implementa il singolo passo del nested-loop: avanza su s2 e, se finisce, avanza su s1; quando non ci sono più record in s1 lo scan è completo, e il metodo ritorna false, vediamo
- I metodi di get controllano dove è presente il campo richiesto, vediamo

ProductScan, qualche dettaglio

```
public class ProductScan implements Scan {
   private Scan s1, s2;
public ProductScan(Scan s1, Scan s2) { …; s1.next(); }
…
public boolean next() {
      if (s2.next()) return true;
      else { s2.beforeFirst();
               return s2.next() && s1.next(); }
   }
public Constant getVal(String fldname) {
      if (s1.hasField(fldname))  return s1.getVal(fldname);
      else return s2.getVal(fldname);
   }
…
public boolean hasField(String fldname) {
      return s1.hasField(fldname) || s2.hasField(fldname);
   }
}
```

Pipelining vs materializzazione

- Due alternative per le interrogazioni nei sottoalberi:
  - **pipelining**: le ennuple sono "utilizzate" dal nodo superiore a mano a mano che vengono prodotte (anzi, vengono prodotte a richiesta):
    - vantaggio: non dovendo salvare i risultati intermedi, riduce i costi di I/O
    - svantaggio: se i risultati intermedi vengono riutilizzati più volte, è necessario ricalcolarli (ad esempio, il ciclo interno di un nested loop)
  - **materializzazione**: l'intero risultato intermedio viene prodotto e memorizzato, prima di essere utilizzato

- Le implementazioni che stiamo vedendo ora sono pipelined

Piano di esecuzione

- Uno scan è l'implementazione di una interrogazione, finalizzata alla sua esecuzione
- Ad essa è associato un **plan** che descrive lo stesso albero, con i costi relativi:
  - Vengono creati diversi plan equivalenti, per confrontare i costi e poi eseguire lo scan più conveniente

L'interfaccia Plan

```
public interface Plan{
    public Scan open(); \* crea lo Scan
    public int blockAccessed();
    public int recordsOutput();
    public int distinctValues(String fldname);
    public Schema schema();
    }
```

Classi che implementano Plan

- Mostriamo i costruttori:

  public TablePlan(String tblname ti, Transaction tx);

  public SelectPlan(Plan p, Predicate Pred);

  public ProjectPlan(Plan p, Collection<String> fldList);

  public ProductPlan(Plan s1, Plan s2);

Costo di uno scan (interessa al plan)

- Quali letture vengono fatte?
- Chi fa la lettura? La pin e dove?
  - Nel costruttore di RecordPage
  - … next() di RecordFile
  - … next() di uno scan

- Quali statistiche servono?
  - Numero di blocchi
  - Numero di record
  - Numero di valori diversi per ciascun campo

Costi e statistiche

  - Numero di blocchi
  - Numero di record
  - Numero di valori diversi per ciascun campo

- TableScan: come nella tabella
- SelectScan: blocchi come nello scan interno, record e valori diversi, sulla base della condizione
- ProjectScan: blocchi come nello scan interno, …
- ProductScan: numero di blocchi della esterna più il prodotto del numero di record restituiti dallo scan esterno per il il numero di blocchi dello scan interno

- Vediamo qualche implementazione (sul codice)

Predicate

- Omettiamo i dettagli. Cenni:
  - Predicato: combinazione booleana di termini (solo and nella versione base)
  - Termine: confronto (=, >, …) fra due espressioni (solo =)
  - Espressione: attributo, costante o espressione (ad esempio aritmetica) su di essi
- Esempio:
  - A = B AND C = 1 — predicato
  - A = B — termine
  - C = 1 — termine
  - C — espressione
  - 1 — espressione

Predicate, 2

- Gli oggetti Predicate sono
  - costruiti dal parser
  - valutati nella SelectScan
  - stimati nel SelectPlan
  - analizzati dal Planner (vedremo più avanti)

Expression

```
public interface Expression {
   public boolean  isConstant(); \* è una costante?
   public boolean  isFieldName(); \* è un nome di attributo?
   public Constant asConstant(); \* restituisce la costante
   public String   asFieldName(); \* restituisce il nome dell'attributo
   public Constant evaluate(Scan s); \* valuta l'espressione sul
                                     \* record corrente dello scan
   public boolean  appliesTo(Schema sch); \* verifica se i campi
                                          \* sono contenuti nello schema
}
public class FieldNameExpression implements Expression {
            public FieldNameExpression(String fldname) { … }
 }
public class ConstantExpression implements Expression {
            public ConstantExpression(Constant c) { … }
}
```

Term

```
public class Term {
    public Term(Expression lhs, Expression rhs)

    public boolean appliesTo(Schema sch)
    public Constant equatesWithConstant(String fldname)
    public String equatesWithField(String fldname)
    public boolean isSatisfied(Scan s)
    public int reductionFactor(Plan p)
}
```

Predicate

```
public class Predicate {
      public class Predicate {
            private List<Term> terms = new ArrayList<Term>();

       public Predicate()
       public Predicate(Term t)

      public void conjoinWith(Predicate pred)
      public boolean isSatisfied(Scan s)
      public int reductionFactor(Plan p)
      public Predicate selectPred(Schema sch)
      public Predicate joinPred(Schema sch1, Schema sch2)
      Constant equatesWithConstant(String fldname)
      String equatesWithField(String fldname)
}
```

SimpleDB, componenti "read-only"

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Planner

- Una classe che contiene due oggetti, con tipi definiti da interfacce
  - un QueryPlanner
  - un UpdatePlanner

Planner e Parser

- Il Planner trasforma un'espressione SQL in un piano di esecuzione e lo esegue:
  - chiama il Parser che gli restituisce un oggetto QueryData, InsertData, etc
  - verifica (con i metadati) che l'oggetto sia semanticamente significativo (p.e., che le tabelle esistano, che abbiano quei campi, che le condizioni …)
  - crea un Plan per l'istruzione SQL
  - a seconda del tipo di operazione
    - per le interrogazioni, restituisce il Plan (serve poi un ResultSet o simile per eseguirlo)
    - per gli aggiornamenti, esegue il Plan

Parser

- Trascuriamo i dettagli
- I metodi fondamentali

  public Parser(String s) /* s è ad esempio la query di interesse

  public QueryData query()

  e simili
- QueryData è una classe per rappresentare query, lo vediamo dal costruttore (cfr SELECT fields FROM tables WHERE pred)

  public QueryData(Collection<String> fields, Collection<String> tables, Predicate pred)
- quindi query() restituisce una rappresentazione di una SELECT e così via

Planner e Parser

- Il Planner trasforma un'espressione SQL in un piano di esecuzione e lo esegue:
  - chiama il Parser che gli restituisce un oggetto QueryData, InsertData, ect
  - verifica (con i metadati) che l'oggetto sia semanticamente significativo (p.e., che le tabelle esistano, che abbiano quei campi, che le condizioni …)
  - crea un Plan per l'istruzione SQL
  - a seconda del tipo di operazione
    - per le interrogazioni, restituisce il Plan (serve poi un ResultSet o simile per eseguirlo)
    - per gli aggiornamenti, esegue il Plan

Verifica

- Per una SELECT o altra istruzione SQL si dovrebbero fare verifiche
  - Le tabelle e i campi esistono nel catalogo
  - I campi non sono ambigui
  - Le azioni sui campi sono coerenti con i relativi tipi
  - Le costanti sono coerenti con i tipi dei campi
- La versione base non fa nessun controllo e solleva eccezione senza diagnostico utile per ogni errore

Planner e Parser

- Il Planner trasforma un'espressione SQL in un piano di esecuzione e lo esegue:
  - chiama il Parser che gli restituisce un oggetto QueryData, InsertData, ect
  - verifica (con i metadati) che l'oggetto sia semanticamente significativo (p.e., che le tabelle esistano, che abbiano quei campi, che le condizioni …)
  - crea un Plan per l'istruzione SQL
  - a seconda del tipo di operazione
    - per le interrogazioni, restituisce il Plan (serve poi un ResultSet o simile per eseguirlo)
    - per gli aggiornamenti, esegue il Plan

Creazione del piano

- La versione base accetta solo istruzioni molto semplici e crea il piano più semplice (e inefficiente)
  - per una SELECT crea il prodotto cartesiano, seguito dalla selezione e infine dalla proiezione

BasicQueryPlanner (senza view)

```
public class BasicQueryPlanner implements QueryPlanner {
public Plan createPlan(QueryData data, Transaction tx) {
    //Step 1: Create a plan for each mentioned table or view
    List<Plan> plans = new ArrayList<Plan>();
    for (String tblname : data.tables()) {
       plans.add(new TablePlan(tblname, tx));
    }

    //Step 2: Create the product of all table plans
    Plan p = plans.remove(0);
    for (Plan nextplan : plans)
      p = new ProductPlan(p, nextplan);

    //Step 3: Add a selection plan for the predicate
    p = new SelectPlan(p, data.pred());

    //Step 4: Project on the field names
    p = new ProjectPlan(p, data.fields());
    return p;
  }
}
```

BasicQueryPlanner

```
public class BasicQueryPlanner implements QueryPlanner {

public Plan createPlan(QueryData data, Transaction tx) {
    //Step 1: Create a plan for each mentioned table or view
    List<Plan> plans = new ArrayList<Plan>();
    for (String tblname : data.tables()) {
      String viewdef = SimpleDB.mdMgr().getViewDef(tblname, tx);
      if (viewdef != null)
        plans.add(SimpleDB.planner().createQueryPlan(viewdef, tx));
      else
        plans.add(new TablePlan(tblname, tx));
    }
    //Step 2: Create the product of all table plans
    Plan p = plans.remove(0);
    for (Plan nextplan : plans)
      p = new ProductPlan(p, nextplan);
    //Step 3: Add a selection plan for the predicate
    p = new SelectPlan(p, data.pred());
    //Step 4: Project on the field names
    p = new ProjectPlan(p, data.fields());
    return p;
  }
}
```

Riassumendo, la classe Planner (per le interrogazioni)

```
public class Planner {
   private QueryPlanner qplanner;
   private UpdatePlanner uplanner;

   public Planner(QueryPlanner qplanner, UpdatePlanner uplanner) { …}

   public Plan createQueryPlan(String qry, Transaction tx) {
      Parser parser = new Parser(qry);
      QueryData data = parser.query();
      // code to verify the query should be here
      return qplanner.createPlan(data, tx);
   }

   public int executeUpdate(String cmd, Transaction tx) {
    …
   }
}
```

- Il Planner viene creato dalla richiesta "remota" e crea qplanner con un costruttore che, nella versione base, usa BasicQueryPlanner() ma può essere sostituito con uno più sofisticato (vedremo più avanti se avremo tempo)

Miglioriamo le prestazioni

- La versione base utilizza un Planner molto rozzo e non sfrutta in alcun modo eventuali strutture fisiche diverse dal file hash
- Miglioramenti
  - Accesso diretto
  - Materializzazione e ordinamento
  - Utilizzo efficace dei buffer
  - "Ottimizzazione" delle interrogazioni

Accesso diretto: indici e hash

- Interfaccia Index, fa riferimento ad un solo campo, permette
  - l'accesso diretto puntuale e la scansione dei record dell'indice (con il valore richiesto): beforeFirst(), next()
  - il riferimento al record puntato dal record corrente dell'indice (un RID è formato da blocco e numero record nel blocco)
  - aggiornamento

```
public interface Index {
   public void    beforeFirst(Constant searchkey);
   public boolean next();
   public RID     getDataRid();
   public void    insert(Constant dataval, RID datarid);
   public void    delete(Constant dataval, RID datarid);
   public void    close();
}
```

Indice, implementazioni

- Le citiamo, senza vederle
  - Hash statico
    - Per esercizio si propone l'hash dinamico
  - B-tree

Utilizzo degli indici

- Implmentazioni degli operatori con utilizzo degli indici
  - Select:
    - IndexSelectScan (e IndexSelectPlan)
  - Join
    - IndexJoinPlan (IndexJoinPlan)

IndexSelectPlan

- È definito su
  - un Plan, come SelectPlan (però si assume che sia un TablePlan)
  - un indice
  - una costante (che associata all'indice, corrisponde ad un predicato semplice, di uguaglianza fra la costante e il campo su cui è definito l'indice)
- Il metodo open() (che come abbiamo visto, deve creare lo scan), apre l'indice e lo passa, insieme alla costante e allo scan sulla tabella, a IndexSelectScan

IndexSelectPlan, variabili e costruttori

```
public class IndexSelectPlan implements Plan {
   private Plan p;
   private IndexInfo ii;
   private Constant val;

   public IndexSelectPlan(Plan p, IndexInfo ii, Constant val,
            Transaction tx) {
      this.p = p;
      this.ii = ii;
      this.val = val;
   }
```

IndexSelectPlan, metodi

```
  public Scan open() {
    // throws an exception if p is not a tableplan.
    TableScan ts = (TableScan) p.open();
    Index idx = ii.open(); // apre l'indice associato a ii
    return new IndexSelectScan(idx, val, ts); // crea e restituisce l'index scan
  }

  public int blocksAccessed() { return ii.blocksAccessed() + recordsOutput(); }

  public int recordsOutput() {   return ii.recordsOutput();

  public int distinctValues(String fldname) {  return ii.distinctValues(fldname); }

  public Schema schema() {  return p.schema(); }
}
```

IndexSelectScan

- Utilizza indice, costante e scan della tabella
  - L'indice ha un record corrente (dell'indice), che alla creazione è subito prima del primo record con la costante sul campo dell'indice
  - Lo scan della tabella un record corrente
  - next() avanza su entrambi e le getXX() operano sul record corrente del file

IndexSelectScan, variabili e costruttore

```
public class IndexSelectScan implements Scan {
   private Index idx;
   private Constant val;
   private TableScan ts;
public IndexSelectScan(Index idx, Constant val, TableScan ts) {
      this.idx = idx;
      this.val = val;
      this.ts  = ts;
      beforeFirst();
   }
```

IndexSelectScan, metodi

```
public void beforeFirst() { idx.beforeFirst(val); }
public boolean next() {
      boolean ok = idx.next();
      if (ok) {
         RID rid = idx.getDataRid();
         ts.moveToRid(rid);
      }
      return ok;
   }
public void close() {
      idx.close();
      ts.close();
   }
public Constant getVal(String fldname) { return ts.getVal(fldname); }
public int getInt(String fldname) {   return ts.getInt(fldname); }
public String getString(String fldname) { return ts.getString(fldname); }
public boolean hasField(String fldname) {  return ts.hasField(fldname);}
```

IndexJoinPlan

- Implementazione dell'operatore di join (prodotto cartesiano seguito da selezione su un predicato p):
  - join(T1, T2, p)
- Si può utilizzare se:
  - T2 è memorizzata
  - P ha la forma "A=B", con A attributo di T1 e B di T2
  - T2 ha un indice sul campo B
- In questo caso, sappiamo che si può utilizzare il nested loop con indice
- Il costruttore usa l'indice su T2 e il nome A per descrivere la condizione di join
- Metodi: al solito, open() e le statistiche (guardare)

IndexJoinPlan, variabili e costruttore

```
public class IndexJoinPlan implements Plan {
   private Plan p1, p2;
   private IndexInfo ii;
   private String joinfield;
   private Schema sch = new Schema();

   public IndexJoinPlan(Plan p1, Plan p2, IndexInfo ii, String joinfield,
                        Transaction tx) {
      this.p1 = p1;
      this.p2 = p2;
      this.ii = ii;
      this.joinfield = joinfield;
      sch.addAll(p1.schema());
      sch.addAll(p2.schema());
   }
```

IndexJoinPlan, metodi

```
public Scan open() {
    Scan s = p1.open();
    // throws an exception if p2 is not a tableplan
    TableScan ts = (TableScan) p2.open();
    Index idx = ii.open();
    return new IndexJoinScan(s, idx, joinfield, ts);
  }
  public int blocksAccessed() {
    return p1.blocksAccessed()
      + (p1.recordsOutput() * ii.blocksAccessed())
      + recordsOutput();
  }
  public int recordsOutput() { return p1.recordsOutput() * ii.recordsOutput(); }
  public int distinctValues(String fldname) {
    if (p1.schema().hasField(fldname))
      return p1.distinctValues(fldname);
    else
      return p2.distinctValues(fldname);
  }
  public Schema schema() { return sch;}
```

IndexJoinScan, variabili e costruttore

```
public class IndexJoinScan implements Scan {
   private Scan s;
   private TableScan ts;  // the data table
   private Index idx;
   private String joinfield;

   public IndexJoinScan(Scan s, Index idx, String joinfield,
            TableScan ts) {
      this.s = s;
      this.idx  = idx;
      this.joinfield = joinfield;
      this.ts = ts;
      beforeFirst();
   }
```

IndexJoinScan, metodi per la scansione

```
public void beforeFirst() {
   s.beforeFirst();
   s.next();
   resetIndex();
}
public boolean next() {
   while (true) {
      if (idx.next()) {
         ts.moveToRid(idx.getDataRid());
         return true;
      }
      if (!s.next())
         return false;
      resetIndex();
   }
}
 private void resetIndex() {
   Constant searchkey = s.getVal(joinfield);
   idx.beforeFirst(searchkey);
}
```

IndexJoinScan

```
public void close() {
    s.close();
    idx.close();
    ts.close();
 }
 public Constant getVal(String fldname) {
    if (ts.hasField(fldname))
       return ts.getVal(fldname);
    else
       return s.getVal(fldname);
 }
 public int getInt(String fldname) {… }
 public String getString(String fldname) { …}
 public boolean hasField(String fldname) {
    return ts.hasField(fldname) || s.hasField(fldname);
 }
```

Materializzazione

- Può essere conveniente materializzare i risultati intermedi, piuttosto che produrli con il pipelining
  - se vengono usati più volte
  - se il costo della materializzazione (che include una rimemorizzazione) e delle successive scansioni è inferiore a quello delle scansioni ripetute
- Costo della materializzazione:
  - costo del piano coinvolto più il numero dei blocchi (per memorizzare)
  - costo della successiva scansione
- In un prodotto cartesiano, può convenire materializzare il secondo argomento (che si legge tante volte), ma non il primo

MaterializePlan

- Una implementazione di Plan che materializza un piano
- Variabili e costruttore

```
public class MaterializePlan implements Plan {
   private Plan srcplan;
   private Transaction tx;

   public MaterializePlan(Plan srcplan, Transaction tx) {
      this.srcplan = srcplan;
      this.tx = tx;
   }
```

MaterializePlan, metodo open esegue

```
public Scan open() {
   Schema sch = srcplan.schema();
   TempTable temp = new TempTable(sch, tx);
   Scan src = srcplan.open();
   UpdateScan dest = temp.open();
   while (src.next()) {
      dest.insert();
      for (String fldname : sch.fields())
         dest.setVal(fldname, src.getVal(fldname));
   }
   src.close();
   dest.beforeFirst();
   return dest;
}
```

MaterializePlan, statistiche

```
public int blocksAccessed() {
    // create a dummy TableInfo object to calculate record length
    TableInfo ti = new TableInfo("", srcplan.schema());
    double rpb = (double) (BLOCK_SIZE / ti.recordLength());
    return (int) Math.ceil(srcplan.recordsOutput() / rpb);
  }

  public int recordsOutput() { return srcplan.recordsOutput(); }

  public int distinctValues(String fldname) {
          return srcplan.distinctValues(fldname);
  }

  public Schema schema() {  return srcplan.schema(); }
}
```

Mergesort

- Va materializzato o quasi (non avrebbe senso ad ogni next() cercare il minimo fra i record non ancora considerati!)
- Extrernal sorting:
  - assume porzioni di file ordinate (ad esempio, singoli blocchi)
  - ad ogni iterazione fonde due o più porzioni consecutive
  - in effetti, l'ultima iterazione non ha bisogno di essere materializzata

- Classi
  - SortPlan
  - SortScan

GroupBy

- Sort (e quindi materializzazione) e aggregazione

MergeJoin

- Come lo conosciamo, richiede relazioni ordinate

Utilizzo dei buffer

- Tanto il sort quanto il prodotto cartesiano (e quindi il join) possono trarre vantaggio dalla disponibilità dei buffer
- Le classi relative possono essere modificate.
  - Per il mergesort, non è difficile, basta far costruire porzioni iniziali ordinate lunghe tanti blocchi quanti sono i buffer disponibili e poi fare un merge a più vie
  - Per il prodotto cartesiano, è necessaria una variante dello scan della tabella, che legga più blocchi, senza rilasciarli (perché altrimenti si eseguirebbe la unpin):
    - classe ChunkScan
    - Classi MultiBufferProductPlan e MultiBufferProductScan (legge un "chunk" di una tabella e scandisce per intero l'altra)

HashJoin

- Pure implementato sfruttando i buffer

Ottimizzazione delle interrogazioni

- L'ottimizzatore di SimpleDB (package opt) procede in due passi:
  - euristiche per costruire l'albero
  - euristiche per scegliere l'implementazione per ciascun nodo

Costruzione dell'albero

- Euristiche
  - Selezioni al più presto
  - Join invece di prodotti cartesiani e selezioni
  - Alberi sbilanciati "left-deep query trees" (i nodi sono join di un sottoalbero con una relazione):
    - In questo modo il sottoalbero destro è materializzato e si possono sfruttare gli indici
  - Negli alberi, si privilegiano i join ai prodotti cartesiani (questi, se ci sono, sono in alto, cioè alla fine)
  - Costruzione aggiungendo una relazione alla volta, scegliendo di volta in volta quella che porta al risultato più piccolo per il join (o prodotto cartesiano) che si sta costruendo

Implementazione dei singoli nodi

- Scelte bottom-up, partendo cioè dalle foglie
- Per ciascun nodo indpendentemente dagli altri
- Euristiche dipendenti dall'operatore:
  - per le select, si valuta se usare l'indice
  - per i join … (la scelta è semplicistica, si dovrebbe usare un'euristica o una valutazione di costo, ma si fa il nested loop con indice, se c'è indice, altrimenti prodotto cartesiano multibuffer)

SimpleDB, componenti per le transazioni

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Il log manager

- Gestisce un file sequenziale, scrivendoci record che il recovery manager (nel modulo transaction) gli passa
- Approccio
  - Mantieni una pagina P allocata per il log
  - Quando viene fornito un nuovo record da inserire nel log
    - Se nella pagina P non c'è posto
      - Scrivi P su disco e reinizializza P
    - Inserisci il nuovo record in P
  - Quando viene richiesta la scrittura su disco di uno specifico record del log
    - Se tale record è in P
      - Scrivi P su disco

Log Manager, API

LogMgr

- public LogMgr(String logfile)
- public void flush(int lsn)
- public synchronized Iterator<BasicLogRecord> iterator()
- public synchronized int append(Object[] rec)

BasicLogRecord

- public BasicLogRecord(Page pg, int pos)
- public int nextInt()
- public String nextString()

LogIterator

- LogIterator(Block blk)
- public boolean hasNext()
- public BasicLogRecord next()

Log Manager

- Il sistema ha un singolo oggetto LogMgr, creato allo startup, associato ad un file (il parametro del costruttore ha il nome)
- append(…) inserisce nel log un record (array di valori, che entri in una pagina; i valori sono interi o stringhe) e restituisce un intero che identifica il blocco nel log (log sequence number, LSN); non garantisce che il record venga scritto su disco
- flush(lsn) garantisce che il record con id lsn e tutti i precedenti vadano su disco
- iterator() restituisce un iteratore di BasicLogRecord (che a sua volta restituisce i record in ordine inverso, partendo dagli ultimi inseriti)

Log Manager, implementazione

- flush scrive su disco il blocco corrente, se necessario; LSN identifica il blocco, quindi la flush riscrive il blocco corrente se la richiesta fa riferimento al suo LSN
- append(…)
  - verifica se c'è spazio nella pagina corrente (calcola la somma delle dimensioni degli oggetti nell'array);
    - in caso negativo scrive la pagina corrente e ne alloca una nuova (metodo privato appendNewBlock());
  - copia gli oggetti (a uno a uno, usando i metodi di Page) e collega fra loro i record del log
  - restituisce il numero del blocco (attraverso un metodo privato)
- iterator() esegue una flush e restituisce LogIterator (un iteratore di BasicLogRecord)

Nota: quanto sono grandi i record del log?

BasicLogRecord

- Permette di leggere un valore alla volta nel record corrente
- Record e valore sono accessibili attraverso una posizione in una pagina
- Non "interpreta" in alcun modo i valori e non sa quanti ce ne siano in un record (se ne occupano le classi che la utilizzano)

SimpleDB, componenti per le transazioni

Remote: riceve le richieste dal client e passa SQL a Planner

Planner: chiama Parser e determina piano e lo passa al Query

Parser: analisi sintattica

Query: riceve il piano e chiama il Record per ogni tabella

Metadata: gestisce gli schemi delle tabelle

Record: gestisce i blocchi per i record delle tabelle

Transaction: gestisce la concorrenza e l'affidabilità

Buffer: mantiene in memoria le pagine per limitare gli accessi

Log: tiene traccia delle operazioni, per l'affidabilità

File: legge e scrive le pagine su disco

Gestione delle transazioni in SimpleDB

- Tre package
  - simpledb.tx (transaction)
  - simpledb.tx.recovery
  - simpledb.tx.concurrency

Transaction, API

- void commit()
- void rollback()
- void recover()

- pin(Block blk)
- unpin(Block blk)
- getInt(Block blk, int offset), getString(…), setInt(…), setString(…)

- int size(String filename)
- Block append(String filename, PageFormatter fmtr)

Gestione dell'affidabilità: Recovery Manager

- Classe principale:
  - RecoveryMgr
- Ogni transazione ha un proprio oggetto RecoveryMgr, che scrive sul log i record opportuni
- Altre classi per gestire i record e per iterare sul log
- API:
  - RecoveryMgr(int txnum) scrive il record di start
  - void commit() scrive il record di commit
  - void rollback() scrive il rec rollback e lo esegue
  - void recover() scrive il rec recover e lo esegue
  - setInt(…), setString(…) scrivono i record di update

I record del log

- Per il log manager sono array di valori, per il recovery manager hanno una struttura e un significato
- Interfaccia
  - LogRecord
- Classi che la implementano (NB: SimpleDB usa una granularità a livello di singoli valori)
  - StartRecord
  - SetIntRecord
  - SetStringRecord
  - CommitRecord
  - RollbackRecord
  - CheckpointRecord

Interfaccia LogRecord

- static final int CHECKPOINT = 0, START = 1 …
  - codifica i tipi di record
- static final LogMgr logMgr = SimpleDB.logMgr();
  - Mantiene un riferimento al log manager
- int writeToLog()
  - scrive il record nel log e restituisce il LSN
- int op()
  - restituisce il tipo di record
- int txNumber()
  - restituisce l'identificatore della transazione
- void undo(int txnum)
  - disfa l'operazione (significativo solo per SetStringRecord e SetIntRecord)

- NB: SimpleDB usa una strategia undo-only

Una classe che implementa LogRecord: SetStringRecord

- Guardare il codice
- Due costruttori:
  - Uno per preparare la scrittura del record nel log
  - L'altro per gestirlo quando viene letto dal log per un rollback o recovery
- Metodi interessanti:
  - int writeToLog()
  - void undo(int txnum)

SetStringRecord, metodi interessanti

```
public int writeToLog() {
    Object[] rec = new Object[] {SETSTRING, txnum,
      blk.fileName(), blk.number(), offset, val};
    return logMgr.append(rec);
  }
 public void undo(int txnum) {
    BufferMgr buffMgr = SimpleDB.bufferMgr();
    Buffer buff = buffMgr.pin(blk);
    buff.setString(offset, val, txnum, -1);
    buffMgr.unpin(buff);
  }
```

LogRecordIterator

- Classe che gestisce l'iterazione (a ritroso) sul log
- È un iteratore di LogRecord
- Utilizza l'iteratore di BasicLogRecord del LogMgr
- Vedere il codice:
  - Il metodo next() ottiene (con una chiamata del relativo next()) un BasicLogRecord dall'iteratore del LogMgr
  - Determina il tipo della registrazione
  - Costruisce il record corrispondente a tale tipo

RecoveryMgr, implementazione

- Un campo privato con l'id della transazione

• RecoveryMgr(int txnum)	scrive il record di start
• void commit()	scrive il record di commit
• void rollback()	esegue rollback e scrive il rec
• void recover()	esegue recover e scrive il rec
• setInt(…), setString(…)	scrivono i record di update

- Utilizza una politica "undo only"
- Non registra le operazioni relative a file temporanei

- Vediamo il codice

RecoveryMgr, costruttore

```
public RecoveryMgr(int txnum) {
    this.txnum = txnum;
    new StartRecord(txnum).writeToLog();
  }
```

RecoveryMgr, commit()

```
public void commit() {
    SimpleDB.bufferMgr().flushAll(txnum);
    int lsn = new CommitRecord(txnum).writeToLog();
    SimpleDB.logMgr().flush(lsn);
  }
```

- La prima operazione è fondamentale per la politica undo-only (garantisce che non servano redo)

RecoveryMgr, rollback()

```
public void rollback() {
      doRollback();
      SimpleDB.bufferMgr().flushAll(txnum);
      int lsn = new RollbackRecord(txnum).writeToLog();
      SimpleDB.logMgr().flush(lsn);
   }
private void doRollback() {
      Iterator<LogRecord> iter = new LogRecordIterator();
      while (iter.hasNext()) {
         LogRecord rec = iter.next();
         if (rec.txNumber() == txnum) {
            if (rec.op() == START)
               return;
            rec.undo(txnum);
         }
      }
   }
```

RecoveryMgr, recover()

```
public void recover() {
    doRecover();
    SimpleDB.bufferMgr().flushAll(txnum);
    int lsn = new CheckpointRecord().writeToLog();
    SimpleDB.logMgr().flush(lsn);
  }

private void doRecover() {
    Collection<Integer> finishedTxs = new ArrayList<Integer>();
    Iterator<LogRecord> iter = new LogRecordIterator();
    while (iter.hasNext()) {
      LogRecord rec = iter.next();
      if (rec.op() == CHECKPOINT)
        return;
      if (rec.op() == COMMIT || rec.op() == ROLLBACK)
        finishedTxs.add(rec.txNumber());
      else if (!finishedTxs.contains(rec.txNumber()))
        rec.undo(txnum);
    }
  }
```

Checkpoint

- Nella versione scaricabile, viene effettuato solo un checkpoint alla fine del recovery (e quindi allo startup)
- La realizzazione del checkpoint periodico viene lasciata come esercizio

RecoveryMgr, gestione di update

```
public int setString(Buffer buff, int offset, String newval) {
    String oldval = buff.getString(offset);
    Block blk = buff.block();
    if (isTempBlock(blk)) \\ i file temporanei non sono gestiti
      return -1;
    else
      return new SetStringRecord(txnum, blk, offset,
          oldval).writeToLog();
  }

public int setInt(Buffer buff, int offset, int newval) …
```

Transaction, API

- void commit()
- void rollback()
- void recover()

- pin(Block blk)
- unpin(Block blk)
- getInt(Block blk, int offset), getString(…), setInt(…), setString(…)

- int size(String filename)
- Block append(String filename, PageFormatter fmtr)

Transaction, implementazione

- Ogni transazione ha
  - Un numero
  - Una lista di pagine di buffer su cui ha avuto il pin
  - Un RecoveryMgr
  - Un ConcurrencyMgr (vedremo più avanti)

Transaction, costruttore

```
public Transaction() {
      txnum       = nextTxNumber();
      recoveryMgr = new RecoveryMgr(txnum);
      concurMgr   = new ConcurrencyMgr();
   }

 private static synchronized int nextTxNumber() {
      nextTxNum++;
      System.out.println("new transaction: " + nextTxNum);
      return nextTxNum;
   }
}
```

Transaction, commit(), rollback() e recover()

```
public void commit() {
    recoveryMgr.commit();
    concurMgr.release();
    myBuffers.unpinAll();
    System.out.println("transaction " + txnum + " committed");
  }
public void rollback() {
    recoveryMgr.rollback();
    concurMgr.release();
    myBuffers.unpinAll();
    System.out.println("transaction " + txnum + " rolled back");
  }
public void recover() {
    SimpleDB.bufferMgr().flushAll(txnum);
    recoveryMgr.recover();
  }
```

Transaction, pin() e unpin()

```
public void pin(Block blk) {
      myBuffers.pin(blk);
   }
  public void unpin(Block blk) {
      myBuffers.unpin(blk);
   }
```

- È poi BufferList che gestisce le effettive richieste al buffer manager

Transaction, setXXX e getXXX

```
public String getString(Block blk, int offset) {
    concurMgr.sLock(blk);
    Buffer buff = myBuffers.getBuffer(blk);
    return buff.getString(offset);
  }

public void setString(Block blk, int offset, String val) {
    concurMgr.xLock(blk);
    Buffer buff = myBuffers.getBuffer(blk);
    int lsn = recoveryMgr.setString(buff, offset, val);
    buff.setString(offset, val, txnum, lsn);
  }
```

Gestione della concorrenza: Concurrency Manager

- Utilizza il lock a due fasi stretto, con una granularità a livello di blocco
- Tre classi
  - ConcurrencyMgr
  - LockTable
  - LockAbortException extends RuntimeException
- Ogni transazione ha il proprio ConcurrencyMgr, mentre c'è una sola LockTable condivisa (variabile statica di ConcurrencyMgr)

LockTable e ConcurrencyMgr, API

LockTable

```
public void sLock(Block blk);
public void xLock(Block blk);
public void unLock(Block blk);
```

ConcurrencyMgr

```
public ConcurrencyMgr(int txnum)
public void sLock(Block blk);
public void xLock(Block blk);
public void unLock(Block blk);
```

- nota: LockTable non conosce le transazioni, ma gestisce i lock di tutte

LockTable

- La tabella è implementata con una mappa, che conta i lock, ma non conosce le transazioni

```
private Map<Block,Integer> locks = new
        HashMap<Block,Integer>();
```

- Valori per l'intero:
  - -1: lock esclusivo
  - n > 0: n lock condivisi

LockTable, metodi privati

```
private boolean hasXlock(Block blk) {
      return getLockVal(blk) < 0;
   }

private boolean hasOtherSLocks(Block blk) {
      return getLockVal(blk) > 1;
  }

 private boolean waitingTooLong(long starttime) {
      return System.currentTimeMillis() - starttime > MAX_TIME;
   }

   private int getLockVal(Block blk) {
      Integer ival = locks.get(blk);
      return (ival == null) ? 0 : ival.intValue();
   }
```

LockTable, richiesta di lock condiviso

```
public synchronized void sLock(Block blk) {
    try {
      long timestamp = System.currentTimeMillis();
      while (hasXlock(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
      if (hasXlock(blk))
            throw new LockAbortException();
      int val = getLockVal(blk);  // will not be negative
      locks.put(blk, val+1);
    }
    catch(InterruptedException e) {
            throw new LockAbortException();
    }
  }
```

LockTable, richiesta di lock esclusivo

```
synchronized void xLock(Block blk) {
    try {// it is invoked after a slock has been obtained
      long timestamp = System.currentTimeMillis();
      while (hasOtherSLocks(blk) && !waitingTooLong(timestamp))
        wait(MAX_TIME);
      if (hasOtherSLocks(blk))
        throw new LockAbortException();
      locks.put(blk, -1);
    }
    catch(InterruptedException e) {
      throw new LockAbortException();
    }
  }
```

LockTable, rilascio di lock

```
synchronized void unlock(Block blk) {
    int val = getLockVal(blk);
    if (val > 1)
      locks.put(blk, val-1);
    else {
      locks.remove(blk);
      notifyAll();
    }
 }
```

Richieste e rilascio di lock, osservazioni

- Il ConcurrencyMgr (vediamo dopo) ottiene il lock condiviso prima di chiedere quello esclusivo
- Se il blocco non è disponibile, il thread va in wait, da cui esce se il lock viene rimosso, a causa della notifyAll() oppure allo scadere del timeout
- La gestione è casuale:
  - i thread in attesa sono riattivati in ordine non definito
  - notifyAll() riattiva tutti i thread sullo stesso oggetto LockTable (quindi quelli per tutti i lock indisponibili), con potenziale inefficienza

ConcurrencyMgr

- Una LockTable condivisa fra tutte le istanze

```
private static LockTable locktbl = new LockTable();
```

- Una mappa con i lock della transazione (solo il tipo, "X" per esclusivo e "S" per condiviso)

```
private Map<Block,String> locks  = new
        HashMap<Block,String>();
```

ConcurrencyMgr

```
public void sLock(Block blk) {
     if (locks.get(blk) == null) {
        locktbl.sLock(blk);
        locks.put(blk, "S");
     }
  }
 public void xLock(Block blk) {
     if (!hasXLock(blk)) {
        sLock(blk);
        locktbl.xLock(blk);
        locks.put(blk, "X");
     }
  }
 private boolean hasXLock(Block blk) {
     String locktype = locks.get(blk);
     return locktype != null && locktype.equals("X");
  }
```

ConcurrencyMgr

```
public void release() {
    for (Block blk : locks.keySet())
      locktbl.unlock(blk);
    locks.clear();
 }
```

Transaction, implementazione, completiamo

- Ogni transazione ha
  - Un numero
  - Una lista di pagine di buffer su cui ha avuto il pin
  - Un RecoveryMgr
  - Un ConcurrencyMgr

Transaction, costruttore

```
public Transaction() {
      txnum       = nextTxNumber();
      recoveryMgr = new RecoveryMgr(txnum);
      concurMgr   = new ConcurrencyMgr();
   }

 private static synchronized int nextTxNumber() {
      nextTxNum++;
      System.out.println("new transaction: " + nextTxNum);
      return nextTxNum;
   }
}
```

Transaction, commit(), rollback() e recover()

```
public void commit() {
    recoveryMgr.commit();
    concurMgr.release();
    myBuffers.unpinAll();
    System.out.println("transaction " + txnum + " committed");
  }
public void rollback() {
    recoveryMgr.rollback();
    concurMgr.release();
    myBuffers.unpinAll();
    System.out.println("transaction " + txnum + " rolled back");
  }
public void recover() {
    SimpleDB.bufferMgr().flushAll(txnum);
    recoveryMgr.recover();
  }
```

Transaction, setXXX e getXXX

```
public String getString(Block blk, int offset) {
    concurMgr.sLock(blk);
    Buffer buff = myBuffers.getBuffer(blk);
    return buff.getString(offset);
  }

public void setString(Block blk, int offset, String val) {
    concurMgr.xLock(blk);
    Buffer buff = myBuffers.getBuffer(blk);
    int lsn = recoveryMgr.setString(buff, offset, val);
    buff.setString(offset, val, txnum, lsn);
  }
```

Stallo (e altri fallimenti)

- Lo stallo viene "ipotizzato" pessimisticamente:
  - Se dopo una certa attesa il lock non viene concesso, viene sollevata un'eccezione (LockAbortException) che si suppone venga catturata dal client (nessun modulo di SimpleDB la gestisce)
- Possibili esercizi (da dettagliare sul sito):
  - Vedremo …

SimpleDB, architettura Client-Server

- SimpleDB può essere utilizzato
  - in forma "embedded"
    - chiamando direttamente i metodi del db (che quindi debbono essere visibili)
  - con approccio client-server
    - chiamando un driver che stabilisce una connessione con un server, cui vengono inviati i comandi SQL per il tramite del driver

Accesso ad un sistema BD embedded

```
import simpledb.planner.Planner;
import simpledb.query.Plan;
import simpledb.query.Scan;
import simpledb.tx.Transaction;

public class SimpleDBEmbeddedTest {

   public static void main(String[] args) {
         SimpleDB.init("studentdb");
         Transaction tx = new Transaction();

         Planner planner = SimpleDB.planner();
         String qry = "select sname, gradyear from student";
         Plan p = planner.createQueryPlan(qry, tx);
         Scan s = p.open();

         while (s.next())
               System.out.println(s.getString("sname") + " " + s.getInt("gradyear"));
         s.close();
         tx.commit();
   }
}
```

Accesso ad un server BD

```
public class SimpleCSTest {
	public static void main(String[] args) {
		Connection conn = null;
		try {
			Driver d = new SimpleDriver();
			String url = "jdbc:simpledb://localhost";
			conn = d.connect(url, null);
			Statement stmt = conn.createStatement();
			String qry = "select sname, gradyear from student";
			ResultSet rs = stmt.executeQuery(qry);
			while (rs.next())
				System.out.println(rs.getString("sname") + " " + rs.getInt("gradyear"));
			rs.close();
			conn.commit();
		}
		catch(SQLException e) {
			e.printStackTrace();
		}
		finally {
			try {
				if (conn != null)
				  conn.close();
			}
			catch (SQLException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Confrontiamo

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

Planner planner = SimpleDB.planner();
String qry = "select sname, gradyear "
      + " from student";
Plan p =
      planner.createQueryPlan(qry, tx);
Scan s = p.open();
while (s.next())
      System.out.println(
            s.getString("sname") +
            " "+s.getInt("gradyear"));
s.close();
tx.commit();
```

```
Driver d = new SimpleDriver();
String url = "jdbc:simpledb://localhost";
Connection conn = d.connect(url, null);
Statement stmt =
            conn.createStatement();
String qry = "select sname, gradyear "
            + " from student";
ResultSet rs = stmt.executeQuery(qry);

while (rs.next())
      System.out.println(
            rs.getString("sname") + " " +
            rs.getInt("gradyear"));
rs.close();
conn.commit();
```

Embedded vs client-server

- Embedded:
  - non permette la condivisione: in particolare, la concorrenza non è gestibile dato che ogni programma opera per conto proprio
  - ha quindi senso solo se:
    - non è possibile connettersi con un server (ad esempio, sistema di navigazione GPS, almeno per ora)
    - c'è un'unica applicazione interessata ai dati (ad esempio, un sistema di acquisizione dati da sensori)
  - possono funzionare bene su macchine locali e poco potenti, rinunciando a funzionalità tipiche delle BD complesse

Implementazione del server SimpleDB

- Due aspetti:
  - Gestione della comunicazione client-server
  - Implementazione dell'interfaccia JDBC

Comunicazione client-server: RMI

- RMI (Remote Method Invocation):
  - API Java per la chiamata di metodi remoti (almeno di un altro processo)
  - Ha lo scopo di rendere trasparente la comunicazione, nascondendo molti dettagli, facendo sembrare locale un oggetto remoto
- Si basa sulla definizione di un insieme di interfacce, per ciascuna delle quali esistono due implementazioni
  - implementazione remota: sul server
  - stub: locale al client, generata automaticamente
- Quando il client chiama (localmente) un metodo sull'oggetto stub, la chiamata viene in effetti inviata al server e viene eseguito il metodo omonimo dell'implementazione remota, che restituisce la risposta

Interfacce remote in SimpleDB

- RemoteDriver
- RemoteConnection
- RemoteStatement
- RemoteResultSet
- RemoteMetaData

- Driver
- Connection
- Statement
- ResultSet
- ResultSetMetaData

- Corrispondono alle interfacce di JDBC, con un sottoinsieme dei metodi
- Estendono l'interfaccia Remote (come richiesto da RMI)
- Lanciano RemoteException (come richiesto da RMI) anziché SQLException (come richiesto da JDBC)

Interfacce remote in SimpleDB, dettaglio

```
public interface RemoteDriver extends Remote {
   public RemoteConnection connect() throws RemoteException;
}
public interface RemoteConnection extends Remote {
   public RemoteStatement createStatement() throws RemoteException;
   public void close() throws RemoteException;
}
public interface RemoteStatement extends Remote {
   public RemoteResultSet executeQuery(String qry) throws RemoteException;
   public int          executeUpdate(String cmd) throws RemoteException;
}
public interface RemoteResultSet extends Remote {
   public boolean next()                 throws RemoteException;
   public int getInt(String fldname)     throws RemoteException;
   public String getString(String fldname) throws RemoteException;
   public RemoteMetaData getMetaData()   throws RemoteException;
   public void close()                   throws RemoteException;
}
public interface RemoteMetaData extends Remote {
   public int    getColumnCount()            throws RemoteException;
…
}
```

Interfacce remote in SimpleDB, utilizzo

- Un client potrebbe utilizzare le interfacce in questo modo (in pratica, non le usiamo direttamente, ma attraverso classi JDBC)

1. RemoteDriver rdvr = …
2. RemoteConnection rconn = rdvr.connect();
3. RemoteStatement rstmt = rconn.createStatement();

1. Vediamo tra poco
2. L'oggetto stub rdvr invia una richiesta alla corrispondente implementazione remota (della stessa interfaccia RemoteDriver), che esegue il metodo connect, che crea un oggetto rconn di tipo RemoteConnection (implementazione remota), uno stub del quale viene inviato al client
3. L'oggetto stub rconn invia una richiesta alla corrispondente implementazione remota (della classe RemoteConnection), che esegue il metodo createStatement, che crea un oggetto RemoteStatement …

Java RMI

- Approfondito (in un quadro organico) nel corso di Architetture Software, accenniamo qui i concetti principali

Il registro RMI

- Una struttura (in effetti, un programma) che permette al server di pubblicare informazioni (in particolare gli oggetti remoti e i loro riferimenti remoti) e al client di ottenere stub per oggetti remoti
- Riferimento remoto:
  - riferimento (univoco in rete) a un oggetto RMI remoto
- Stub:
  - Oggetto proxy (rappresentante) di un oggetto RMI remoto
- Quando il client interroga il registro, ottiene un oggetto (stub) che contiene, nel proprio stato, un riferimento remoto (più precisamente, ottiene dal registro il riferimento remoto e la libreria RMI crea lo stub nel processo del client)

Il registro RMI, utilizzo in SimpleDB

- SimpleDB pubblica un oggetto, di tipo RemoteDriver;
  nel main della classe simpledb.server.Startup abbiamo:

```
RemoteDriver d = new RemoteDriverImpl();
Naming.rebind("simpledb", d);
```

- Naming.rebind (nel package java.rmi) associa nel registro l'oggetto remoto d (implementazione remota del driver) ad un nome simbolico (in questo caso "simpledb"), in modo tale che i client possano trovare un oggetto remoto conoscendone il nome simbolico
- Nella classe SimpleDriver abbiamo

```
String newurl = url.replace("jdbc:simpledb", "rmi") +
                                "/simpledb";
RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newurl);
```

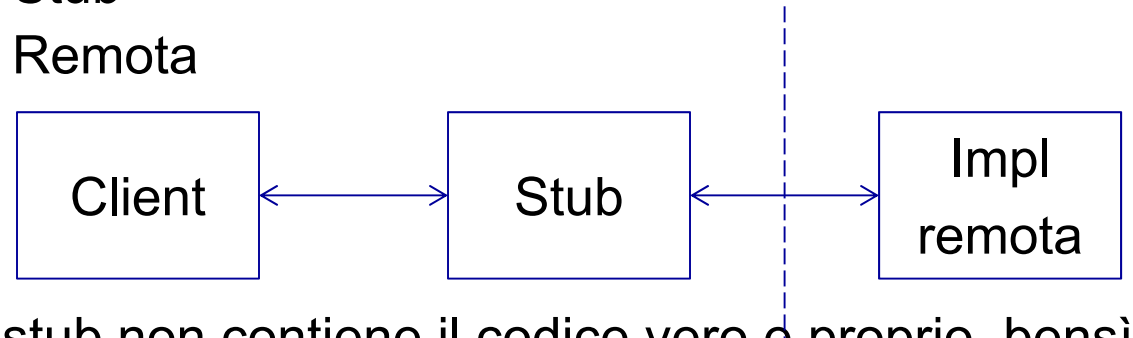
- Naming.lookup cerca nel registro un oggetto remoto a partire dal suo nome simbolico e ne restituisce lo stub

Macchine (virtuali), processi e thread

- In una esecuzione client-server (di SimpleDB, ma anche più in generale) abbiamo due processi (ognuno eventualmente con più thread):
  - un processo per il DBMS sulla macchina server
  - un processo per ciascun client sulla macchina client
- Se è tutto su una macchina fisica, sono comunque due macchine virtuali diverse (in Windows, possiamo verificarlo guardando i processi con il task manager)
- Gli oggetti remoti (nel nostro caso, il RemoteDriver) sono in attesa di richieste, che possono arrivare loro dagli stub: in tal caso, eseguono i metodi richiesti, restituiscono i risultati e tornano ad aspettare
- Dualmente, lo stub, fatta la chiamata remota, aspetta la risposta (restando bloccato)

Implementazione delle interfacce remote

- Per ogni interfaccia remota, ci sono due implementazioni
  - Stub
  - Remota



- Lo stub non contiene il codice vero e proprio, bensì la gestione della comunicazione con l'implementazione remota e per questo viene sostanzialmente generato dalle librerie runtime di RMI
- Nota:
  - Per il driver, lo stub viene ottenuto a partire dal registro RMI
  - Per le altre classi, dal risultato di un metodo dell'implementazione remota che viene trasformato in stub

Interfacce e classi in simpledb.remote

- cinque interfacce
  - RemoteDriver, RemoteConnection, RemoteStatement, RemoteResultSet, RemoteMetaData
- cinque implementazioni remote (delle interfacce)
  - RemoteDriverImpl, RemoteConnectionImpl, RemoteStatementImpl, RemoteResultSetImpl, RemoteMetaDataImpl
- cinque adapter (implementano, in astratto, le interfacce JDBC)
  - DriverAdapter, ConnectionAdapter, StatementAdapter, ResultSetAdapter, ResultSetMetaDataAdapter
- cinque classi per il client JDBC (estendono gli adapter e incartano gli stub delle implementazioni)
  - SimpleDriver, SimpleConnection, SimpleStatement, SimpleResultSet, SimpleMetaData

RemoteDriverImpl

```
package simpledb.remote;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class RemoteDriverImpl extends UnicastRemoteObject
                                      implements RemoteDriver {
  public RemoteDriverImpl() throws RemoteException {
  }

public RemoteConnection connect() throws RemoteException {
    return new RemoteConnectionImpl();
  }
}
```

RemoteDriverImpl, specificità

```
public class RemoteDriverImpl extends UnicastRemoteObject
                                        implements RemoteDriver {
  public RemoteDriverImpl() throws RemoteException {
  }
  public RemoteConnection connect() throws RemoteException {
    return new RemoteConnectionImpl();
  }
}
```

- Il punto di accesso al server: un solo oggetto, creato all'avvio da simpledb.server.Startup
- La chiamata a connect() crea sul server un oggetto RemoteConnectionImpl; RMI (con il codice della classe UnicastRemoteObject) restituisce al client il corrispondente stub, gestendo la comunicazione

RemoteConnectionImpl, metodi nell'interfaccia remota

```
class RemoteConnectionImpl extends UnicastRemoteObject implements
RemoteConnection {
        private Transaction tx;
        RemoteConnectionImpl() throws RemoteException {
                tx = new Transaction();
        }
        public RemoteStatement createStatement()
                                throws RemoteException {
                return new RemoteStatementImpl(this);
        }
        public void close() throws RemoteException {
                tx.commit();
        }
…
}
```

- Un oggetto (su client e su server) per ogni connessione
- Delega praticamente tutto agli oggetti Transaction e RemoteStatementImpl

RemoteConnectionImpl, metodi di package

```
class RemoteConnectionImpl extends UnicastRemoteObject
                                        implements RemoteConnection {
…
Transaction getTransaction() {
      return tx;
 }
void commit() {
      tx.commit();
      tx = new Transaction();
}
void rollback() {
      tx.rollback();
      tx = new Transaction();
}
```

- Sono utilizzati dalle classi RemoteStatementImpl e RemoteResultSetImpl (e non sono visibili per i client)

RemoteStatementImpl, 1

```
class RemoteStatementImpl … {
  private RemoteConnectionImpl rconn;
  public RemoteStatementImpl(RemoteConnectionImpl rconn)
          throws RemoteException {
    this.rconn = rconn;
  }
```

- Fa riferimento alla connessione
- Esegue le istruzioni SQL, con executeQuery() ed executeUpdate(), vediamo

RemoteStatementImpl, executeQuery()

```
public RemoteResultSet executeQuery(String qry) throws RemoteException {
    try {
      Transaction tx = rconn.getTransaction();
      Plan pln = SimpleDB.planner().createQueryPlan(qry, tx);
      return new RemoteResultSetImpl(pln, rconn);
    }
    catch(RuntimeException e) {
      rconn.rollback();
      throw e;
    }
  }
```

- Ottiene un Plan per l'interrogazione (dal planner) e restituisce un RemoteResultSet
- In caso di eccezione fa il rollback della transazione e risolleva la stessa eccezione

RemoteStatementImpl, executeUpdate()

```
  public int executeUpdate(String cmd) throws RemoteException {
    try {
      Transaction tx = rconn.getTransaction();
      int result = SimpleDB.planner().executeUpdate(cmd, tx);
      rconn.commit();
      return result;
    }
    catch(RuntimeException e) {
      rconn.rollback();
      throw e;
    }
  }
}
```

- Fa eseguire l'update al planner (ottenendo e restuendo un codice di stato)
- Implementa la semantica "autocommit"
- Eccezione come prima

RemoteResultSetImpl,1

```
class RemoteResultSetImpl extends UnicastRemoteObject implements RemoteResultSet {
  private Scan s;
  private Schema sch;
  private RemoteConnectionImpl rconn;
  public RemoteResultSetImpl(Plan plan, RemoteConnectionImpl rconn) throws RemoteException {
    s = plan.open();
    sch = plan.schema();
    this.rconn = rconn;
  }
  public boolean next() throws RemoteException {
    try {
      return s.next();
    }
    catch(RuntimeException e) {
      rconn.rollback();
      throw e;
    }
  }
```

- Il costruttore apre il Plan ed estrae Scan e Schema; tiene traccia della connessione
- Gli altri metodi chiamano i metodi corrispondenti di Scan e gestiscono le eccezioni

RemoteResultSetImpl, 2

```
public int getInt(String fldname) throws RemoteException {
  try {
                fldname = fldname.toLowerCase(); // to ensure case-insensitivity
                return s.getInt(fldname);
  }
  catch(RuntimeException e) {
    rconn.rollback();
    throw e;
  }
}
public String getString(String fldname) throws RemoteException {
  ….
}
public RemoteMetaData getMetaData() throws RemoteException {
  return new RemoteMetaDataImpl(sch);
}
public void close() throws RemoteException {
  s.close();
  rconn.commit();
}
```

- getString() richiama il metodo omonimo di Scan
- close() esegue il commit (semantica autocommit)

RemoteMetadataImpl

- Gestisce tipi e proprietà delle colonne di un oggetto RemoteResultSetImpl
- Tralasciamo

Implementazione delle interfacce JDBC

- Le implementazioni delle classi remote
  - sollevano eccezioni RemoteException, come richiesto da RMI
  - Implementano solo alcuni dei metodi di JDBC
- Per renderle compatibili con JDBC è necessario inserirle in classi wrapper che implementino le opportune eccezioni e gestiscano i metodi non implementati
  - catturano le eccezioni remote e sollevano quelle SQL
  - sono definite come sottoclassi di classi adapter, classi astratte che implementano tutti i metodi, ma senza fare niente (o sollevando eccezioni)

DriverAdapter

```
public abstract class DriverAdapter implements Driver {
  public boolean acceptsURL(String url) throws SQLException {
    throw new SQLException("operation not implemented");
  }
  public Connection connect(String url, Properties info) throws SQLException {
    throw new SQLException("operation not implemented");
  }
  public int getMajorVersion() {
    return 0;
  }
  public int getMinorVersion() {
    return 0;
  }
  public DriverPropertyInfo[] getPropertyInfo(String url, Properties info) {
    return null;
  }
  public boolean jdbcCompliant() {
    return false;
  }
}
```

SimpleDriver

```
package simpledb.remote;

import java.sql.*;
import java.rmi.*;
import java.util.Properties;

public class SimpleDriver extends DriverAdapter {

public Connection connect(String url, Properties prop) throws SQLException {
      try {
         String newurl = url.replace("jdbc:simpledb", "rmi") + "/simpledb";
         RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newurl);
         RemoteConnection rconn = rdvr.connect();
         return new SimpleConnection(rconn);
      }
      catch (Exception e) {
         throw new SQLException(e);
      }
   }
}
```

ConnectionAdapter

```
public abstract class ConnectionAdapter implements Connection {
  public void clearWarnings() throws SQLException {
    throw new SQLException("operation not implemented");
  }
  public void close() throws SQLException {
    throw new SQLException("operation not implemented");
  }
  public void commit() throws SQLException {
    throw new SQLException("Commit operation not implemented");
  }
  ...
  public Blob createBlob() throws SQLException {
    throw new SQLException("operation not implemented");
  }
  ...
  public Statement createStatement() throws SQLException {
    throw new SQLException("operation not implemented");
  }
  ...
  public void setAutoCommit(boolean autoCommit) throws SQLException {
    throw new SQLException("operation not implemented");
  }
  ...
}
```

- Circa 50 metodi (non implementati)

SimpleConnection

```
public class SimpleConnection extends ConnectionAdapter {
  private RemoteConnection rconn;
  public SimpleConnection(RemoteConnection c) {
    rconn = c;
  }
  public Statement createStatement() throws SQLException {
    try {
      RemoteStatement rstmt = rconn.createStatement();
      return new SimpleStatement(rstmt);
    }
    catch(Exception e) {
      throw new SQLException(e);
    }
  }
  public void close() throws SQLException {
    try {
      rconn.close();
    }
    catch(Exception e) {
      throw new SQLException(e);
    }
  }
}
```

- Due metodi e un costruttore (tutto utilizzando RemoteConnection)

Ancora

- StatementAdapter e SimpleStatement
- ResultSetAdapter e SimpleResultSet
- ResultSetMetaDataAdapter e SimpleResultSetMetaData

Riassumendo

- cinque interfacce
  - RemoteDriver, RemoteConnection, RemoteStatement, RemoteResultSet, RemoteMetaData
- cinque implementazioni remote (delle interfacce)
  - RemoteDriverImpl, RemoteConnectionImpl, RemoteStatementImpl, RemoteResultSetImpl, RemoteMetaDataImpl
- cinque adapter (implementano, in astratto, le interfacce JDBC)
  - DriverAdapter, ConnectionAdapter, StatementAdapter, ResultSetAdapter, ResultSetMetaDataAdapter
- cinque classi per il client JDBC (estendono gli adapter e incartano gli stub delle implementazioni)
  - SimpleDriver, SimpleConnection, SimpleStatement, SimpleResultSet, SimpleMetaData