EtlTest

Un'applicazione di esempio
per strutturare in pochi passi
un semplice data warehouse
e
la sua interfaccia di navigazione

Introduzione

In questa guida vengono presentati i passi necessari per realizzare una semplice migrazione da un database relazionale ad un data warehouse e la navigazione dello stesso attraverso una semplice interfaccia web.

Il linguaggio di programmazione utilizzato è Ruby[1] e il framework è Ruby on Rails[2]. Si utilizzano inoltre il plugin Activewarehouse[3] e la gemma Activewarehouse-etl[4] all'interno dell'applicazione.

In questa guida si prevede che sia già installato il framework Ruby on Rails e si utilizzi come database MySql. Come ulteriore settaggio iniziale è richiesta l'istallazione della gemma Activewarehouse-etl[5].

---

1 Ruby 1.8.6
2 Rails 2.2.2
3 Active Warehouse 0.4.0
4 Active Warehouse-etl 0.9.1
5 In ambiente linux con il comando `>sudo gem install activewarehouse-etl`

Definizione schema

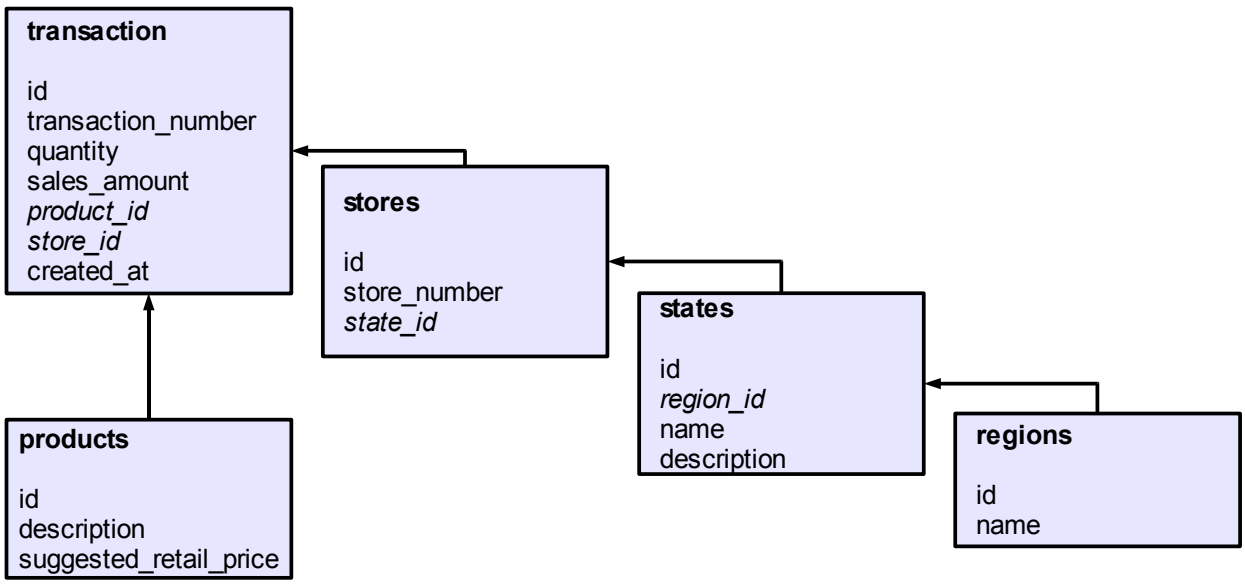
In questa guida si ripercorrono le tappe che portano all'acquisizione dei dati provenienti da un database organizzato in modo relazionale e il caricamento degli stessi in un data warehouse opportunamente strutturato.

L'evoluzione delle relazioni e la trasformazione della strutturazione delle tabelle e dei campi si può notare confrontando i due schemi sintetici delle tabelle dei rispettivi db.

Il database relazionale è formato da cinque tabelle collegate tra loro come compare nello schema sottostante. Le voci in corsivo sono i campi ponte che permettono di correlare le differenti tabelle fra loro permettendo di andare a recuperare le informazioni contenute nelle altre tabelle attraverso il valore dell'indice dei singoli record (*id*).

La tabella *Transaction* contiene un insieme di operazioni di vendita identificate da un numero identificativo (*transaction_number*), la quantità di prodotto venduta (*quantity*), l'importo complessivo della transazione (*sales_amount*), il riferimento al prodotto venduto (*product_id*) e al negozio nel quale la transazione ha avuto luogo (*store_id*) ed infine la data in cui tale transazione ha avuto luogo (*created_at*).

La tabella *Products* contiene una semplice descrizione del prodotto (*description*) e il prezzo suggerito di vendita (*suggested_retail_price*). Le tre tabelle *Stores*, *States* e *Regions*, collegate tra loro a cascata, contengono rispettivamente le informazioni riguardanti il numero identificativo del negozio e il riferimento allo Stato in cui è situato (*stores.strore_number* e *stores.state_id*), la sigla dello Stato, il suo nome per esteso e il riferimento alla regione degli Stati Uniti di cui fa parte (*states.name*, *states.description*, *states.region_id*) ed nell'ultima il nome delle quattro regioni in cui sono suddivisi (*regions.name*).

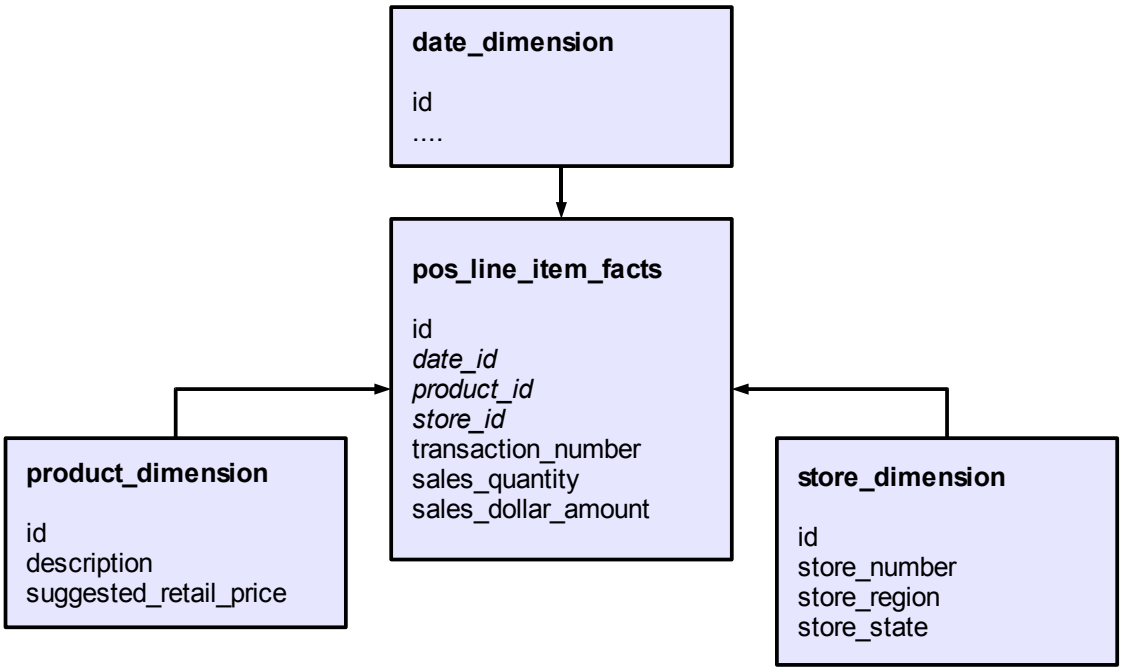


Tale strutturazione dei dati permette di ridurre la possibilità di errori in quanto ogni informazione è memorizzata in un unico record impedendo la presenza di informazione ripetute.

Volendo rendere possibile la navigazione dei dati in modo semplice e intuitivo permettendo di aggregare e disaggregare i dati raccolti secondo criteri (le dimensioni) e livelli di aggregazione differenti (le gerarchie) è necessario ristrutturare il database. In particolare esplicitare i criteri di analisi, chiamate dimensioni, mettendoli in relazione diretta con i dati memorizzati che contengono le informazioni legate ai fatti in analisi. Tale scelta porta a strutturare il data warehouse "a stella" avendo come centro focale i fatti in analisi e come "raggi" le dimensioni di analisi.

Lo schema ramificato a più livelli del database relazione e quello tipicamente a stella del data warehouse permettono di cogliere immediatamente quali siano le operazioni da prevedere nel corso dell'operazione di estrazione dei dati.

In particolare si possono notare l'astrazione della data e l'accorpamento nella "store_dimension" delle informazioni racchiuse nelle tabelle "stores", "states" e "regions".



Ottenuto un data warehouse così strutturato è utile introdurre il concetto di "cubo" che permette di esprimere in maniera chiara quali siano i dati che si intendono analizzare (i fatti in analisi) e quali siano le dimensioni, ovvero i criteri attraverso cui filtrare i dati, e le gerarchie in base ai quali aggregarli.

In questa guida in particolare seguiremo i passi necessari alla definizioni e al popolamento della dimensione "tempo" (*date_dimension*), che permette di aggregare e disaggregare in base a criteri nidificati come mese, settimana, giorno della settimana, della dimensione "negozio" (*store_dimension*) che permette di navigare i dati in base al negozio in cui la compravendita è stata effettuata e al suo posizionamento geografico ed infine la dimensione "prodotto" (*product_dimension*) che nel nostro caso prevede una gerarchia ad un solo livello.

La dimensione temporale in particolare richiede che la data espressa in forma esplicita nel database relazionale nel campo *created_at*, venga rielaborata come informazione al pari delle altre creando una tabella apposita[6]. In tal modo nel campo *date_id* del data warehouse sarà presente solamente un numero intero corrispondente al record della tabella *date_dimension* che rappresenta un determinato giorno.

6 Cfr R. Kimball, The Data Warehouse Toolkit, (pag 38 ss.)

Creazione Data warehouse

Dopo la definizione dello schema bisogna costruire definendole le tabelle che si sono individuate precedentemente.

Creazione applicazione e database

Per prima cosa si crea l'applicazione "etlTest" che gestirà il data warehouse (l'opzione `-d mysql` segnala a rails di voler utilizzare mysql come database).

```
> rails etlTest -d mysql
```

Si modifica il file di configurazione[7] dei database aggiungendo un ulteriore database ai tre ambienti tipici di ogni applicazione realizzata con RoR.

Questa aggiunta è necessaria per permettere alla gemma che gestisce l'etl di lavorare.

Si crea quindi il db con l'apposita funzione

```
> cd etlTest
/etlTest> rake db:create:all
```

```
defaults: &defaults
  adapter: mysql
  encoding: utf8
  username: root
  password: <password>
  socket: /var/run/mysqld/mysqld.sock

development:
  database: etlTest_development
  <<: *defaults

test:
  database: etlTest_test
  <<: *defaults

production:
  database: etlTest_production
  <<: *defaults

etl_execution:
  database: etl_execution
  <<: *defaults
```

File 1: etlTest/config/database.yml

Si installa inoltre il plugin ActiveWarehouse

```
/etlTest> script/plugin install  git://github.com/aeden/activewarehouse.git
```

Creazione dimensione tempo standard

Come prima operazione è quindi necessaria la creazione della dimensione "tempo", sfruttando il plugin appena installato eseguiamo[8]

```
/etlTest> script/generate dimension date
```

Che genera i seguenti file

- create app/models/date_dimension.rb
- create test/unit/date_dimension_test.rb
- create test/fixtures/date_dimension.yml
- create db/migrate
- create db/migrate/20090325153810_create_date_dimension.rb[9]

tralasciando i file predisposti per i test utilizzeremo in primo luogo il file per la migration che integreremo seguendo l'esempio del File 2 prima di lanciare i due comandi per creare la tabella e quindi effettuare il popolamento dei dati.

7 Cfr. File 1: etlTest/config/database.yml

8 Ricordiamo che è necessaria la gemma adapter_extention che viene già automaticamente installata con l'installazione di ActiveWarehouse-etl

9 Il nome del file per la migration varia in base all'istante in cui viene creato essendo generato secondo lo schema: aaaammgghhmmss_create_date_dimension.rb

```
/etlTest> rake db:migrate
/etlTest> rake warehouse:build_date_dimension
```

Andando ad ispezionare la tabella *date_dimension* nel database *etlTest_development* troveremo un elenco di record corrispondenti ognuno ad un giorno a partire da cinque anni prima la data attuale. La funzione rake *warehouse:build_date_dimension* definisce per ogni giorno anche tutte le altre variabili scelte come il nome del mese, il numero dell'anno, il giorno della settimana...

ActiveWarehouse prevede infatti l'utilizzo, se lo si desidera, di un formato data standardizzato[10] per il quale è prevista l'utile funzione rake che permette di popolare la tabella della dimensione tempo in modo automatico. Tale popolamento permette di generare in modo automatizzato tutte le informazioni necessarie per poter utilizzare i vari livelli di aggregazione disponibili.

Il plugin si aspetta infatti di trovare una tabella adeguatamente formattata che popolerà di default con la serie completa dei giorni da cinque anni prima il giorno dell'esecuzione del comando.

L'arco temporale è modificabile a piacere passando come riferimenti START_DATE e END_DATE per indicare un intervallo diverso da quello predefinito. Con questo comando[11] ad esempio viene popolata la tabella *date_dimension,* rimuovendo i dati presenti precedentemente, cominciando dal 28 agosto 1988 fino al 31 gennaio 2010.

```
/etlTest> rake START_DATE='08/28/1988' END_DATE='01/31/2010' TRUNCATE=true
                                                    warehouse:build_date_dimension
```

La definizione dei campi che si intendono creare viene gestita in modo semplice e intuitivo commentando o decommentando i nomi proposti (mettere o togliere # all'inizio della riga) per avere a disposizione quel campo. Nel caso in cui si volesse inserire un ulteriore campo basta modificare opportunamente la classe di popolamento ActiveWarehouse::Builder::DateDimensionBuilder del plugin installato[12] e inserire il nome del campo nell'elenco del File 2.

Costruzione struttura tabella warehouse

Dopo aver definito la dimensione temporale si devono generare le altre due previste dallo schema: *store* e *product.*

Utilizziamo lo script di generazione automatica che creerà i file[13] per le migration nella directory */db/migrate/* e i file di controllo[14] che modificheremo successivamente in */app/models.*

```
/etlTest> script/generate dimension store
/etlTest> script/generate dimension product
```

Si aggiungono quindi i campi individuati precedentemente nella definizione dello schema aggiungendoli nell'apposito campo fields (File 3) e similmente si esegue per la tabella *product_dimension* (File 4).

Si segue al stesa procedura anche per la tabella dei fatti inserendo i campi già individuati all'inizio

```
/etlTest> script/generate fact pos_line_item
```

con la differenza che questi non verranno indicizzati (File 5).

10 Si utilizzano come campi quelli proposti da Kimball nell'opera citata precedentemente; il file per la migration è reperibile on-line all'indirizzo http://activewarehouse.rubyforge.org/svn/demo/trunk/db/migrate/001_create_date_dimension.rb

11 Ovviamente l'istruzione è su una sola riga

12 All'interno del file etlTest/vendor/plugins/activewarehouse/lib/active_warehouse/builder/date_dimension_builder.rb

13 Nella cartella etlTest/db/migrate/
20090327132651_create_store_dimension.rb
20090327132821_create_product_dimension.rb

14 Nella cartella etlTest/app/models/
store_dimension.rb
product_dimension.rb

Definite quindi tutte le istruzioni per la creazione della struttura del data warehouse lo creiamo

```
/etlTest> rake db:migrate
```

```
class CreateDateDimension < ActiveRecord::Migration
  def self.up
    fields = {
      :date => :string,
      :full_date_description => :text,
      :day_of_week => :string,
      #:day_number_in_epoch => :integer,
      #:week_number_in_epoch => :integer,
      #:month_number_in_epoch => :integer,
      :day_number_in_calendar_month => :integer,
      :day_number_in_calendar_year => :integer,
      :day_number_in_fiscal_month => :integer,
      :day_number_in_fiscal_year => :integer,
      #:last_day_in_week_indicator => :string,
      #:last_day_in_month_indicator => :string,
      :calendar_week => :string,
      #:calendar_week_ending_date => :string,
      :calendar_week_number_in_year => :integer,
      :calendar_month_name => :string,
      :calendar_month_number_in_year => :integer,
      :calendar_year_month => :string,
      :calendar_quarter => :string,
      :calendar_year_quarter => :string,
      #:calendar_half_year => :string,
      :calendar_year => :string,
      :fiscal_week => :string,
      :fiscal_week_number_in_year => :integer,
      :fiscal_year_month => :string,
      :fiscal_quarter => :string,
      :fiscal_year_quarter => :string,
      #:fiscal_half_year => :string,
      :fiscal_year => :string,
      :holiday_indicator => :string,
      :weekday_indicator => :string,
      :selling_season => :string,
      :major_event => :string,
      :sql_date_stamp => :date
    }

    create_table :date_dimension do |t|
      fields.each do |name,type|
        t.column name, type
      end
    end
    fields.each do |name,type|
     add_index :date_dimension, name unless type ==
:text
    end

  end

  def self.down
    drop_table :date_dimension
  end
end
```

File 2: etlTest/db/migrate/20090325153810_create_date_dimension.rb

```
class CreateStoreDimension < ActiveRecord::Migration
  def self.up
    fields = {
      :id => :integer,
      :store_number => :string,
      :store_region => :string,
      :store_state => :string
    }
    create_table :store_dimension do |t|
      fields.each do |name,type|
        t.column name, type
      end
    end
    fields.each do |name,type|
      add_index :store_dimension, name unless type == :text
    end
  end

  def self.down
    drop_table :store_dimension
  end
end
```

File 3: etlTest/db/migrate/20090327132651_create_store_dimension.rb

```
class CreateProductDimension < ActiveRecord::Migration
  def self.up
    fields = {
      :id => :integer,
      :description => :string,
      :suggested_retail_price => :float
    }
    create_table :product_dimension do |t|
      fields.each do |name,type|
        t.column name, type
      end
    end
    fields.each do |name,type|
      add_index :product_dimension, name unless type == :text
    end
  end

  def self.down
    drop_table :store_dimension
  end
end
```

File 4: etlTest/db/migrate/20090327132741_create_product_dimension.rb

```
class CreatePosLineItemFacts < ActiveRecord::Migration
  def self.up
    create_table :pos_line_item_facts do |t|
      t.integer :id
      t.integer :date_id
      t.integer :product_id
      t.integer :store_id
      t.string  :transaction_number
      t.integer :sales_quantity
      t.float   :sales_dollar_amount
    end
  end

  def self.down
    drop_table :pos_line_item_facts
  end
end
```

File 5: etlTest/db/migrate/20090327140840_create_pos_line_item_facts.rb

Operazioni Etl

Con l'acronimo Etl si raggruppano i processi *Extract*, *Transform* e *Load* con i quali i dati dal loro supporto originale vengono opportunamente immagazzinati in un sistema di sintesi (data warehouse, data mart, ...).

L'insieme delle operazioni di trasformazione ha lo scopo di rendere omogenei dati anche provenienti da fonti diverse e strutturarli nella forma più opportuna per lo scopo che ci si pone (in ambito aziendale alla logica di business per cui viene sviluppato).

La principale caratteristica che discrimina la qualità dell'operazione è la scelta della granularità dei dati che risulta essere un compromesso tra la possibilità di effettuare analisi approfondite e l'attenzione al rischio della decadenza delle prestazioni legata all'eccesso di specificazione.

ActiveWarehouse-etl è una gemma che permette di gestire questo processo utilizzando un semplice file in cui, seguendo apposite convenzioni, si definiscono tutti i passi necessari in maniera trasparente ed intuitiva.

In particolare all'interno di questi file con terminazione .ctl si possono individuare sei porzioni di codice, non necessariamente tutte presenti:

- *Local function*: si definiscono funzioni particolari richiamabili all'interno del file o i riferimenti a librerie o file esterni
- *Define source*: definizione riferimenti a file dati esterni sia di ingresso, sia di uscita, e ai file di configurazione dei database con cui si interagisce
- *Extract*: le istruzioni per le estrazioni dati nella forma `source :name, configuration, definition`
- *Transform*: le operazioni di elaborazione sui dati `transform(name, value, row)`
- *Load*: generazione dati in uscita nella forma `destination :name, configuration, definition`
- *Post process*: operazioni eseguite dopo la scrittura dei dati in uscita di tipo bulk import o encode

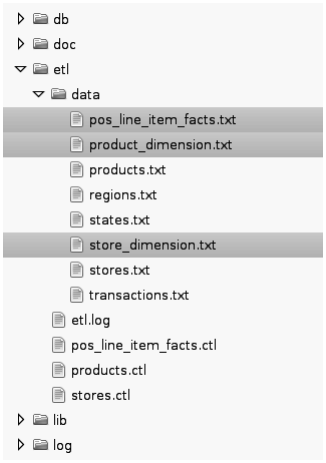
in aggiunta è possibile utilizzare *Screens* per inserire controlli di coerenza a vari livelli di rilevanza prima che i dati arrivino in produzione.

Fonti dati

I dati a cui si fa riferimento sono recuperabili all'indirizzo

http://activewarehouse.rubyforge.org/svn/presentations/trunk/data/

Questo è un esempio in cui i dati vengono recuperati da file di testo di tipo txt, ma è ovviamente possibile avere come fonte file di tipo diverso, database transazionali, fogli di calcolo, file di log.



Posizionamento dati e file

I file ctl di controllo e quelli contenenti i dati del nostro database di partenza abbiamo scelto di memorizzarli all'interno dell'applicazione cercando di mantenere un certo ordine. I primi nella directory *etlTest/etl*, i secondi nella sottodirectory *etlTest/etl/data*.

I file evidenziati nell'immagine sono quelli generati dall'esecuzione dei processi di elaborazione, similmente al file di log, anch'esso generato automaticamente.

Esecuzione Etl

Scaricati i dati e definiti i file di controllo (File 6, 7, 8) questi vengono eseguiti con i comandi

```
/etlTest> etl etl/products.ctl
/etlTest> etl etl/stores.ctl
/etlTest> etl etl/pos_line_item_facts.ctl
```

Unitamente ad eventuali messaggi di errore la gemma fornisce anche alcuni dati statistici sulla velocità con cui le operazioni sono state svolte.

L'ordine di esecuzione degli script dipende dalla necessità per le transazioni degli id che individuano i negozi.

Products.ctl

Per prima cosa vengono definiti i riferimenti alla fonte dati *infile* e al file in cui i dati elaborati verranno memorizzati *outfile* indicandone il posizionamento.

Quindi la fonte dati (*:file*), il criterio per parsarli (*:parser*) e l'opzione di saltare la prima riga, e quindi i nomi dei campi che verranno utilizzati per riferirsi ad essi [:id, ...].

In questo caso non sono previste trasformazioni particolari, ma i dati così strutturati vengono esportati secondo le istruzioni seguenti la chiave *destination* specificando l'ordine con cui farlo attraverso i nomi campi.

Fase finale il caricamento da *outfile* al database attraverso *:bulk_import* a cui vengono passati gli adeguati parametri.

:file è il file da cui trarre i dati.

:truncate è un parametro in base al quale esegue o meno il troncamento della tabella aggiungendo così i dati nella tabella svuotata o in coda a quelli già esistenti

:columns è l'elenco dei campi da riempire e *:field_separator* il carattere che distingue nel file di orgine il contenuto di una cella dalla vicina.

:target è l'ambiente specificato nel file di configurazione dell'applicazione rails all'interno del quale andare a cercare la tabella indicata dal parametro *:table*.

```
infile = 'data/products.txt'
outfile= 'data/product_dimension.txt'

source :in, {
  :file => infile,
  :parser => :delimited,
  :skip_lines => 1
},
[
  :id,
  :description,
  :suggested_retail_price
]

destination :out, {
  :file => outfile,
  :separator => "\t"
},
{
  :order => [:id, :description,
             :suggested_retail_price]
}

post_process :bulk_import, {
  :file => outfile,
  :truncate => true,
  :columns => [:id,:description,
               :suggested_retail_price],
  :field_separator => "\t",
  :target => :development,
  :table => 'product_dimension'
}
```

File 6: etltest/elt/products.ctl

L'utilizzo del passaggio intermedio con il file *product_dimension.txt* non è tecnicamente necessario in quanto sarebbe possibile caricare direttamente i dati sul db.

La separazione in due fasi è però consigliata per poter inserire eventuali test di verifica prima dell'effettiva messa in produzione ad esecuzione delle trasformazioni avvenuta.

Il file *outfile* permette inoltre un facile controllo dei risultati e la possibilità di effettuare più importazioni parziali in tabelle diverse.

```
class StoreParser < ETL::Parser::DelimitedParser
  include Enumerable
  def initialize(source)
    @parser = ETL::Parser::DelimitedParser.new(source)
    super
  end

  def each
    @parser.each do |store|
      row = {}
      row[:id] = store[:id]
      row[:store_number] = store[:store_number]
      state = lookup_state(store[:state_id])
      row[:store_state] = state[3]
      region = lookup_region(state[1])
      row[:store_region] = region[1]
      yield row
    end
  end

  private
  def lookup_region(id)
    #puts "lookup region #{id}"
    @regions ||= load_regions
    @regions.each do |region|
      return region if region[0] == id
    end
    raise ArgumentError, "Region not found for id #{id}"
  end
  def load_regions
    regions = FasterCSV.read(File.dirname(__FILE__) + "/data/regions.txt")
    regions.shift
    regions
  end
  def lookup_state(id)
    @states ||= load_states
    @states.each do |state|
      return state if state[0] == id
    end
    raise ArgumentError, "State not found for id #{id}"
  end
  def load_states
    states = FasterCSV.read(File.dirname(__FILE__) + "/data/states.txt")
    states.shift
    states
  end
end

infile = 'data/stores.txt'
outfile = 'data/store_dimension.txt'

source :in, {  :file => infile,
               :parser => StoreParser,
               :skip_lines => 1           },
            [  :id,
               :state_id,
               :store_number              ]

destination :out, {  :file => outfile,
                     :separator => "\t"   },
                  {
                       :order => [:id, :store_number, :store_region, :store_state]
                  }

 post_process :bulk_import, {
   :file => outfile,
   :truncate => true,
   :columns => [:id, :store_number, :store_region, :store_state],
   :field_separator => "\t",
   :target => :development,
   :table => 'store_dimension'
 }
```

File 7: etltest/etl/stores.ctl

Stores.ctl

Nel passaggio dalla struttura ramificata, tipica dei database relazionali, a quella a stella, le tre tabelle che permettevano di descrivere gli *stores* devono essere compressi in un unica tabella con tre campi.

Una delle possibili vie è quella presentata (File 7) in cui si definisce una classe locale che estende le tipologie di parser disponibili. Questa estensione definisce la nuova procedura *StoreParser* che richiama la dichiarazione del *source* e genera i tre campi necessari a schiacciare l'informazione. In particolare ripercorre ricorsivamente i dati delle tre tabelle originali in modo da generarne una sola.

Questo esempio rende evidente la libertà con cui in ogni caso è possibile operare con i file ctl in quanto rimangono dei file in cui viene eseguito del codice ruby. È quindi possibile inserire funzioni o qualsiasi altra funzionalità in modo intuitivo e lineare.

Pos_line_item_facts.ctl

Rispetto agli altri processi vi sono alcuni passaggi ulteriori.

In primo luogo viene incluso tutto l'ambiente tramite il *require* iniziale che permette di chiamare direttamente la classe *DateDimension* nella fase di trasformazione.

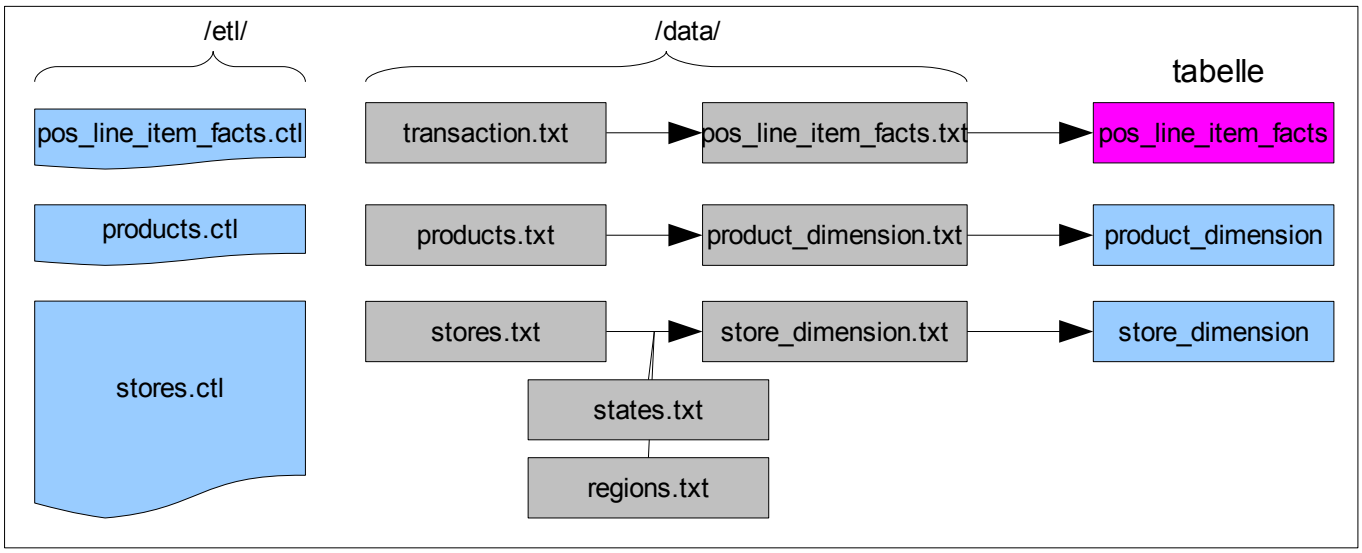
```
require 'config/environment'
infile = 'data/transactions.txt'
outfile = 'data/pos_line_item_facts.txt'
set_error_threshold 50

source :in, {
  :file => infile,
  :parser => :delimited,
  :skip_lines => 1
},
[
  :id, :transaction_number, :qty, :sales_amount,
  :product_id, :store_id, :date_id
]

transform :date_id, :string_to_date
transform :date_id, :foreign_key_lookup,
       {:resolver => ActiveRecordResolver.new(DateDimension,
                                    :find_by_sql_date_stamp)}

destination :out, {
  :file => outfile,
  :separator => "\t"
},
{
  :order => [:id, :date_id, :product_id, :store_id,
             :transaction_number, :sales_quantity,
             :sales_dollar_amount]
}

post_process :bulk_import, {
  :file => outfile,
  :truncate => true,
  :columns => [:id, :date_id, :product_id, :store_id,
               :transaction_number, :sales_quantity,
               :sales_dollar_amount],
  :field_separator => "\t",
  :target => :development,
  :table => 'pos_line_item_facts'
}
```

File 8: etlTest/etl/pos_line_item_fact.ctl

Compare quindi la chiamata *transform* che in due passaggi passa dalla stringa letta nel file txt al riferimento alla tabella *date_dimension*.

Il primo è la conversione dalla stringa di testo ad una data.

Il secondo il rimpiazzo della stessa con l'id corrispondente al giorno indicato. Tale id viene ricavato dall'interrogazione della tabella *date_dimension* precedentemente popolata con la funzione predisposta *:find_by_sql_date_stamp.*



Definizione dimensioni fatti e cubi

Importati quindi i dati non rimane che definire quali siano le aggregazioni e confronti che interessino per avere a disposizione il data warehouse funzionante.

Dimensioni

Primo passo è la definizione delle gerarchie che si vogliono rendere disponibili per la navigazione dei dati. I livelli indicati in ogni gerarchia individuano un livello di aggregazione e l'ordine in cui sono presentati definisce il loro annidiamento. La definizione di più gerarchie permette la specificazione di percorsi di analisi diversi richiamabili nella fase di renderizzazione dei dati.

Nel caso specifico sono state individuate due gerarchie per la dimensione temporale di cui una a quattro livelli di aggregazione.

```
class DateDimension < ActiveWarehouse::Dimension
  set_order :sql_date_stamp
  define_hierarchy :calendar_year, [:calendar_year, :calendar_month_name,
                                    :calendar_week, :day_of_week]
  define_hierarchy :day_of_week,   [:day_of_week]
end
```

File 9: etlTest/app/models/date_dimension.rb

Questo vuol dire che se verrà scelta la gerarchia *calendar_year* i dati saranno disponibili in forma aggregata per anno, mese dell'anno, settimana del mese, giorni della settimana. Mentre nel caso della gerarchia *day_of_week* saranno disponibili tutte le vendite di tutti gli anni presenti nel db raggruppate per giorno della settimana.

Per l'analisi delle dimensioni *store* e *product* ne è stata indicata solo una ciascuna.

```
class ProductDimension < ActiveWarehouse::Dimension
  define_hierarchy :description, [:description]
end
```

File 10: etlTest/app/models/product_dimension.rb

```
class StoreDimension < ActiveWarehouse::Dimension
  define_hierarchy :location, [:store_region, :store_state]
end
```

File 11: etlTest/app/models/store_dimension.rb

Nel caso in una dimensione siano specificate più gerarchie si ritiene predefinita quella scritta per prima. Ovvero, nel caso non sia indicato esplicitamente un'altra gerarchia, quella utilizzata sarà la prima decritta.

Quindi nel caso venga utilizzata la dimensione DateDimension e non sia indicata esplicitamente una gerarchia sarà usata di default *calendar_year*.

Fatti

```
class PosLineItemFact < ActiveWarehouse::Fact
  aggregate :id,            :type => :count,  :label => "Purchases"

  dimension :date
  dimension :store
  dimension :product

  belongs_to :date,     :class_name => "DataDimension"
  belongs_to :store,    :class_name => "StoreDimension"
  belongs_to :product,  :class_name => "ProductDimension"
end
```

File 12: etlTest/app/models/pos_line_item_fact.rb

Nella definizione del model che si riferisce ai fatti (File 12) bisogna indicare eventuali campi da calcolare appositamente dal data warehouse non presetni nei dati, ma da essi ricavabili, le dimensioni lungo le quali i fatti sono esplorabili ovvero con cui sono in relazione e la specificazione del tipo di relazione esistente.

Aggregate è un esempio di creazione di un campo ottenuto contando (*:type => :count*) il numero di occorrenze del campo *:id* e assegnandoli il label Purchase.

Sono quindi indicate le tre dimensioni in base alle quali questi fatti possono essere analizzati e la relazione, uno a molti (*belong_to*) esistente tra i fatti e i record della tabella corrispondente alla dimensione.

Cubi

Una volta generati e descritti nelle loro componenti fatti e dimensioni si possono definire i cubi ovvero quali siano le dimensioni di analisi e i fatti su cui si vogliono ottenere i dati aggregati.

```
class StoreSalesCube < ActiveWarehouse::Cube
  reports_on :pos_line_item
  pivots_on :date, :store
end
```

File 13: etlTest/app/models/store_sales_cube.rb

Per prima cosa si genera il cubo con il comando

```
/etlTest> script/generate cube StoreSales
```

che oltre ai file per i test genera il file nella cartella model *store_sales_cube.rb*.

La definizione dei cubi (File 13) richiede semplicemente l'indicazione delle dimensioni a cui si riferisce con la chiamata pivots_on e dei fatti con la chiamata reports_on.

Realizzazione interfaccia

Seguendo il paradigma Model – View – Controller richiesto dal framework Ruby on Rails si deve creare un file controller in cui specificare i riferimenti ai cubi e alle dimensioni che si intendono navigare.

La scelta fatta è quella per una pagina con i link ai cubi che sfruttano le due differenti gerarchie esplicitate per la dimensione temporale.

Controller

Sfruttando gli script di generazione creiamo i file necessari

```
/etlTest> script/generate controller home
```

Specifichiamo i cubi, i fatti e le dimensioni scelte tramite le semplici dichiarazioni visibili nel listato (File 14) indicando quale dimensione si voglia sulle righe e quale sulle colonne.

Vista la convenzione sulla scelta delle gerarchie nel nostro caso *sales_report* utilizzerà la gerarchia *:calendar_year*, non essendo specificato diversamente, mentre *day_report* la indicata *:day_of_week*.

La chiamata tramite il metodo *.view* permette di aggiornare di volta in volta il set di dati da visualizzare nella rispettiva vista passando come parametri gli indicatori del livello di aggregazione che si intende visualizzare.

```
class HomeController < ApplicationController

  def index
  end

  def sales_report
    @report = ActiveWarehouse::Report::TableReport.new
    @report.cube_name = :store_sales_cube
    @report.column_dimension_name = :date
    @report.row_dimension_name = :store

    @view = @report.view(params)
  end

  def day_report
    @report = ActiveWarehouse::Report::TableReport.new
    @report.cube_name = :store_sales_cube
    @report.column_dimension_name = :date
    @report.row_dimension_name = :store
    @report.column_hierarchy = :day_of_week

    @view = @report.view(params)
  end
end
```

File 14: etlTest/app/controllers/home_controller.rb

Views

Dall'unico file home_controller.rb dipendono quindi le tre viste e un partial:

```
<h1>ActiveWarehouse Examples</h1>

<ul>
  <li><%= link_to 'Sales Report', :action => 'sales_report' %></li>
  <li><%= link_to 'Day Report', :action => 'day_report' %></li>
</ul>
```

File 15: etlTest/app/views/home/index.html.erb

index.html.erb che contiene semplicemente i link alla visualizzazione delle due possibili gerarchie degli stessi dati (File 15).

```
        <h1>Day report</h1>

<%= render :partial => 'table' %>
```

File 16: etlTest/app/views/ home/day_report.html.erb

```
     <h1> Sales Report</h1>

<%= render :partial => 'table' %>
```

File 17: etlTest/app/views/ home/sales_report.html.erb

day_report.html.erb che aggrega le vendite a secondo del giorno della settimana in cui sono stati effettuati(File 16).

sales_report.html.erb che permette la navigazione delle vendite analizzate per negozio di vendita e periodo dell'anno (File 17).

Visto che il codice delle due viste è simile rispettando il principio Don't Repeat Yourself utilizziamo il partial *_table.html.erb* e nel loro codice utilizziamo la funzione `<%= render :partial => 'table' %>`

per richiamare il partial passando i corretti parametri ottenuti dal file controller.

_table.html.erb in cui la chiamata *render_report_from* permette la generazione della tabella mentre i link dinamici ai vari livelli navigati sono resi disponibili dalle due chiamate *render_crumbs* (File 18).

```
<p><%= render_crumbs(@view.column_crumbs) %></p>
<p><%= render_crumbs(@view.row_crumbs) %></p>

<%= render_report_from(@view)%>
```

File 18: etlTest/app/views/home/_table.html.erb

Routes

Per rendere raggiungibile la home con i link alle due viste costruite aggiungiamo nel file di configurazione (File 19) la semplice istruzione

```
map.root, :controller => "home"
```

e rinominiamo o rimuoviamo il file index.html.erb posizionato nella cartella etltest/public.

```
ActionController::Routing::Routes.draw do |map|
  map.root :controller => "home"

  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
```

File 19: etlTest/app/config/routes.rb

In tale maniera al lancio del server saremo direttamente reindirizati correttamente che collega automaticamente

Utilizzo applicazione

Inizializzazione warehouse

Ora che tutto è pronto possiamo lanciare il comando di rake che inizializza il data warehouse popolando tutti i dati aggregati richiesti dal data warehouse. ActiveWarehouse crea le tabelle necessarie e le popola automaticamente in base alle direttive specificate nei file di configurazione dei cubi.

```
/etlTest> rake warehouse:rebuild
```

Questo comando è necessario ogni qual volta si modifichino le informazioni contenute nei file della cartella models.

Navigazione

Per poter quindi navigare nei dati lanciamo in locale il server sulla porta 3000

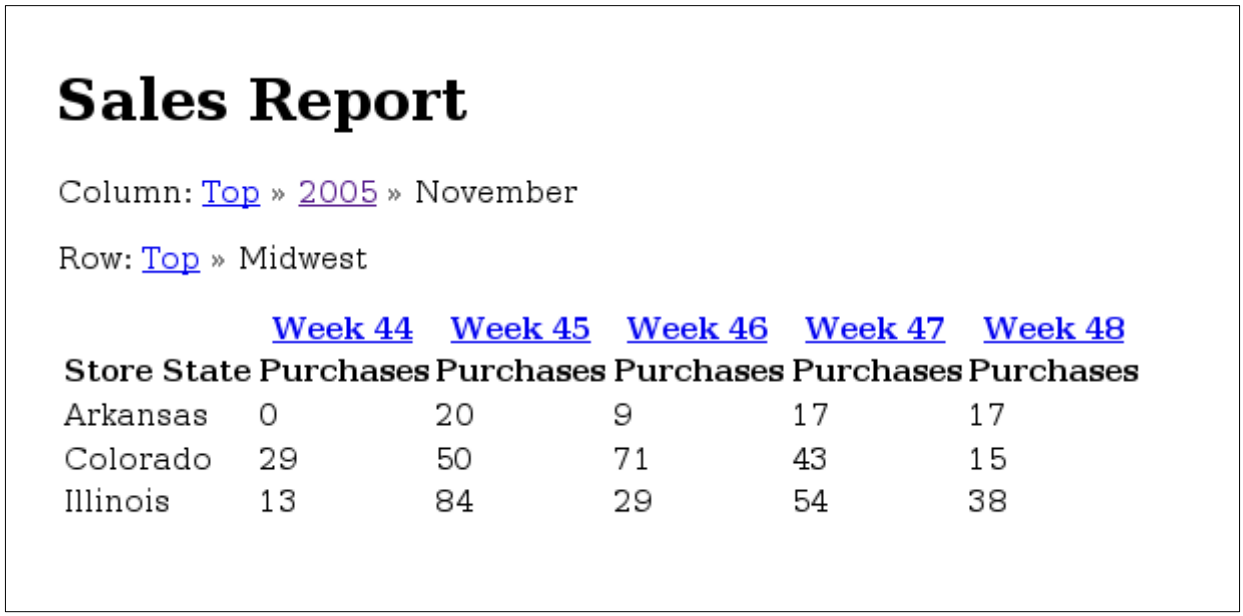
```
/etlTest> script/server -p 3000
```

e con un qualsiasi browser ci rechiamo all'indirizzo

http://localhost:3000/

dove, cliccando su uno dei link, otterremo la semplice interfaccia base che visualizza i dati in formato aggregato.

La navigazione degli stessi si effettua cliccando sui nomi delle colonne per effettuare uno zoom verso il dettaglio, mentre sfruttando i link presenti in capo alla pagine si torna ad un livello di maggiore aggregazione.



Sales Report

Column: Top » 2005 » November

Row: Top » Midwest

	Week 44	Week 45	Week 46	Week 47	Week 48
Store State	Purchases	Purchases	Purchases	Purchases	Purchases
Arkansas	0	20	9	17	17
Colorado	29	50	71	43	15
Illinois	13	84	29	54	38