Data management for Big Data

Professore: Massimo Franceschet

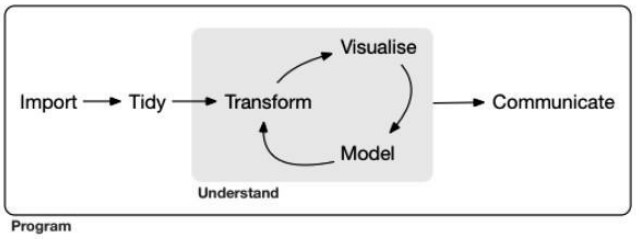
Presentazione:
Nome cognome
Cosa abbiamo studiato
Perché siamo qui

Portare sempre il computer, le lezioni non dureranno sempre 4 ore. Ha il suo sito http://users.dimi.uniud.it/~massimo.franceschet/

1. Introduzione

L'obiettivo è quello di preparare ed analizzare i piccoli dati, non quelli grandi, perché dobbiamo essere in grado di gestire i dati piccoli per fare quelli grandi. Quindi il 90% sarà su "piccoli" dati. Incontreremo in questo percorso Giorgia Lupi, molto interessante sull'aspetto della visualizzazione del dato e Edgar Frank Codd. Il libro che seguiremo è "R for Data Science". L'autore è il creatore di alcuni pacchetti che useremo in questo corso.
Datacamp è una piattaforma di learning di corsi di data science e proveremo ad usarla per gli esercizi con la possibilità di creare una classe per avere accesso gratuito per seguire delle lezioni.

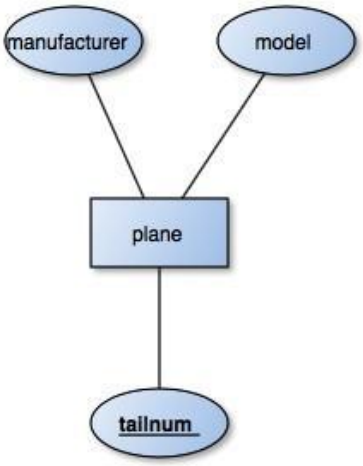
Quello presentato in figura è il ciclo che applicheremo ai nostri dati. Di per sé non c'è una sequenza tra queste fasi. La fase ultima è la **comunicazione**, che secondo il professore è la fase più importante per comunicare i risultati in modo efficace ed esteticamente appagante. La presentazione dei dati solitamente prende 10 minuti alle persone dopo un lavoro di ore da parte dello studioso.



Molti dei dati non sono in forma tabellare, ma sono dati che possono essere rappresentati tramite reti o grafi. Poi, quindi, si tratterà di gerarchie (alberi) con linguaggi XLM e XPATH. Vedremo anche in questo caso le difficoltà nel tabellare i dati ''ambito del text mining.
Un'altra forma di dati sono quelli che arrivano in tempo reale (umidità, movimento, ecc.), useremo due nuovi strumenti Processing e Arduino per analizzare tali dati. Nell'ultima parte del corso studieremo la comunicazione e collaborazione.
Il materiale è sul sito, dove ci sono i link ad ogni slide. Qui c'è il libro di testo: http://r4ds.had.co.nz/introduction.html. Qui ci sono le slide: http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/syllabus.html
Un altro libro molto interessante uscito recentemente Modern data Science with R. Di cui ne leggeremo un due capitoli. Ci sono tutti i libri comunque nel link precedente. In realtà i libri utili sono il primo e il terzo della lista
Sul sito ci sono anche i software da scaricare che utilizzeremo durante il corso. L'80% delle volte useremo R.

l'esame è scritto più un progetto non opzionale. L'esame scritto non è nient'altro che rispondere a delle domande e fare degli esercizi e poi dovremo fare un progetto, cioè risolvere una data challenge. Il progetto è singolare. Nel report non va il codice, verrà inviato separatamente. Il materiale deve essere spedito una settimana prima dell'esame. Lo scritto è sia teorico che pratico (tipo un semplice algoritmo).

2. DataBase Relazionali



Analizzeremo i **database relazionali** che sono i più diffusi. Esistono DB di altra natura, non relazionali. Questi però sono i più diffusi e quelli più collegati con data science (DS). È molto interessante vedere storicamente DB e come si può fare la manipolazione dei dati con una classica relazione e come queste cose sitraducano nel linguaggi della DS. L'argomento DB è molto ampio, servirebbero 9 crediti. Noi cerchiamo di distillare il 15% utile per i nostri studi. Fondamentalmente vedremo quattro cose:

- Modello concettuale
- Modello logico
- Linguaggio per creare il modello fisico e anche per interrogare una base di dati
- Brevissimo confronto tra DB e Data Science.

Useremo **SQLite**, quindi un **database management system**, che gestisce una buona quantità di dati che molti scienziati del dato usano.

E vedremo come è possibile usare dei pacchetti in R per usare i database direttamente dall'ambiente R.

Una **base di dati** è un'insieme di dati tra loro collegati, e sarà possibile da R studiare la relazione dei dati del DB.

2.1 Modello Entity-Relationship

Il **modello relazionale** è un **modello concettuale** che è un modello ad alto livello. Sotto il modello concettuale c'è il **modello logico** e sotto il **modello fisico**, secondo una scala di astrazione. Quindi è il modello più astratto e meno tecnico. In questo modello l'obiettivo è quello di descrivere la realtà che ci appare indipendentemente da come questa realtà venga poi archiviata nella nostra macchina. Ci sono diversi modelli concettuali, noi useremo il modello **Entità Relazioni ER**. Il modello ER definisce uno schema concettuale dei dati. Uno **schema** descrive la struttura o la forma dei dati, ma non dice nulla dei dati (occorrenze, determinazioni, ecc.) Un esempio è lo schema dei voli che potrebbe essere una tabella che contiene le variabili: aeroplani, il modello dell'aeroplano, il costruttore dell'aeroplano. Queste potrebbero essere delle caratteristiche dell'entità volo. Ma non dice i valori assunti da queste variabili. In altre parole lo schema sta all'istanza come la classe sta all'oggetto oppure come il progetto della casa sta alla realizzazione.

Quando parleremo di modello concettuale ci riferiremo allo schema concettuale. Parleremo solo a livello concettuale quelle che sono le entità interessanti e le relazioni che ci sono tra queste entità. Per fare un modello concettuale dobbiamo decidere due cose:

- **Entità**: rappresentano un complesso e rilevante concetto con esistenza indipendente. Una istanza di una entità è un oggetto rappresentato della classe. Per esempio, le possibili entità in un database di voli sono: voli, aeroplani, aeroporti, condizioni meteo.
- **Relazioni**: rappresentano un legame logico, significante per la realtà modellata, tra due entità. Un'istanza di una relazione è una coppia di istanze di due entità che partecipano alla relazione. Per esempio, ci

potrebbe essere una relazione chiamata *flies* tra i voli e gli aerei che associa ogni volo con il corrispondente aereo (o viceversa).

Dobbiamo imparare ad isolare quelli che sono i concetti importanti e complessi e a capire quali sono le proprietà di questi oggetti. Quindi dobbiamo capire quali sono le variabili che sono importanti ai nostri scopi e poi li si disegnano come nel grafico qua sotto.

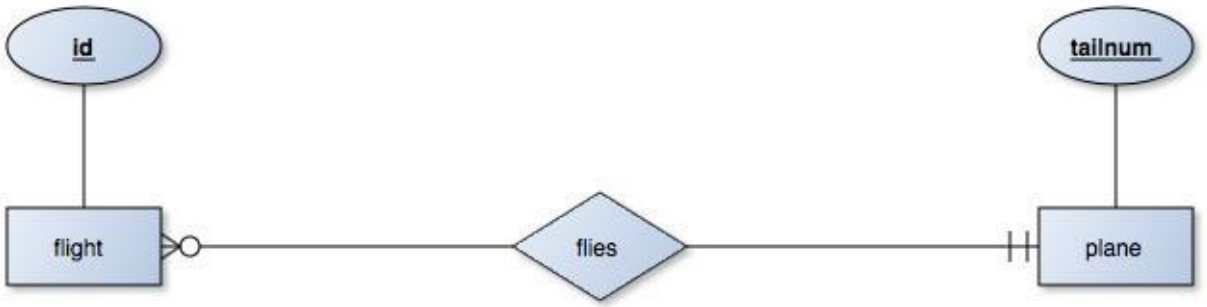
Sia le entità che le relazioni possono avere proprietà atomiche chiamate **attributi**. Questi possono essere **obbligatori** (il valore deve essere dato) o **opzionali** (il valore può non esistere o esser sconosciuto). L'attributo dovrebbe avere un nome univoco per essere riconosciuto.

L'attributo che è sottolineato è una **chiave**, cioè un "codice" che identifica univocamente l'elemento. Una chiave deve avere due requisiti:

- Deve essere **obbligatoria**, cioè non posso lasciarla vuota
- Deve essere **minimale**, cioè è la più piccola possibile. Se prendiamo gli studenti di Trieste ci basterebbe la matricola, se prendiamo anche gli studenti di Udine magari le chiavi si ripetono, quindi ci servirebbe anche un codice per il comune. Potremmo aggiungere anche l'età, ma non sarebbe più minimale come chiave, perché aggiungiamo un attributo inutile all'identificazione. Una chiave è un insieme di attributi che identifica univocamente l'istanza dell'identità.

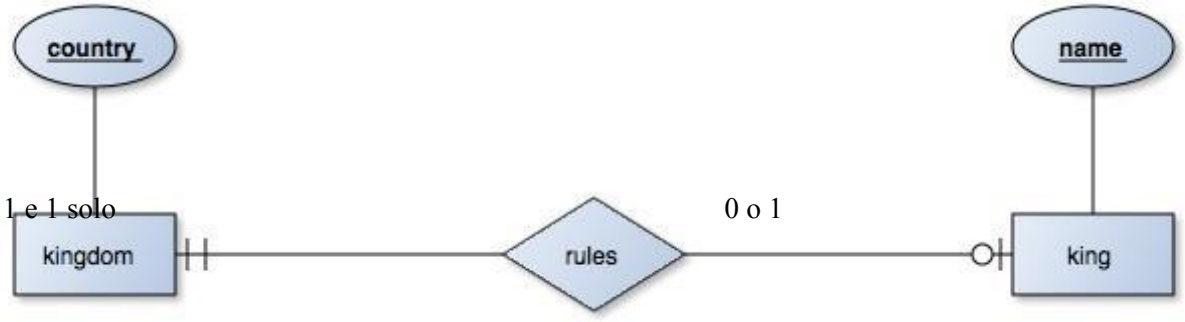
Il secondo concetto quindi sono le relazioni. Che rappresentano un legame logico significativo ed importante per la realtà che andiamo a modellare tra due entità. Due perché vediamo solo relazioni binarie. Questa scelta è dovuta dal fatto che sono le più frequenti, pur esistendo anche con più entità.

Per esempio ci può essere una relazione tra i voli e gli aeroplani. Quindi dobbiamo legare queste due entità. Le relazioni ci servono a questo. Come vediamo questo è un ambiente molto potente, perché in termini di entità e relazioni si possono tradurre molte situazioni.

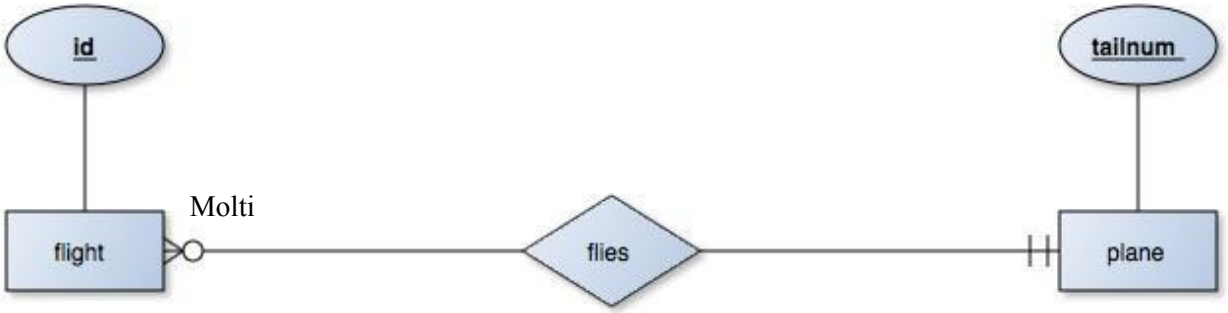


La relazione *flies* è definita con il rombo, ci sono 3 tipi di relazioni:

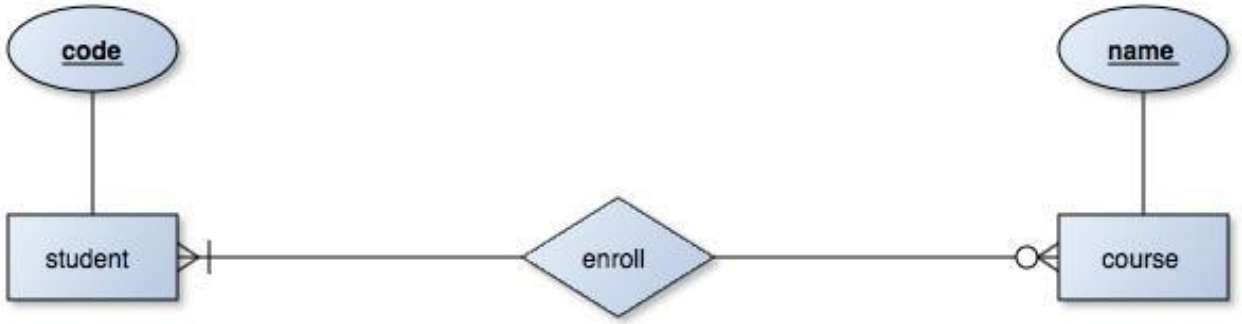
- **Relazione uno a uno**, sostanzialmente una funzione. Come se per esempio abbiamo un regno ed un re, la relazione è "governa", in questo caso ogni re governa un regno e ogni regno è governato da un re, quindi non posso avere regni con più re o un re con più regni. Quindi è una relazione uno a uno. Se ci pensiamo è una funzione $f(x) = y$, cioè ad ogni relazione della x si ha un valore della y.



- **Relazioni uno a molti**, un'istanza di un'entità può essere associata a molte istanze, ma non viceversa.

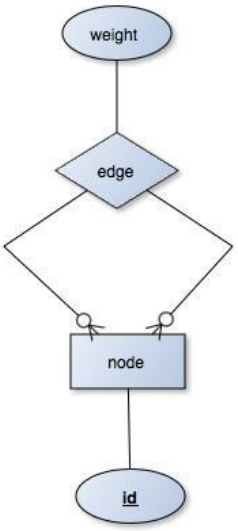


• **Relazione molti a molti:** un'istanza di un'entità può essere associata a più istanze e viceversa. L'esempio classico è relazione tra studenti e corso.

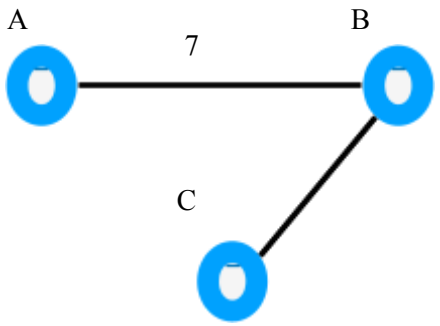


Questi vincoli li poniamo noi, c'è una fase preliminare chiamata **fase dei requisiti**, in qui decido tutte queste caratteristiche. Sembrano concetti banali, ma il tutto è molto potente e serio, se si comprendono bene questi tre concetti possiamo modellare molte situazioni, e possiamo capire molto bene quali sono i requisiti del sistema e tradurli in queste 3 entità.

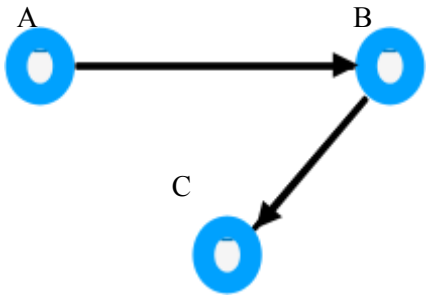
Nulla vieta di aver **relazioni ricorsive**, cioè relazioni che collegano due entità uguali. Una rete si può modellare con una tabella e una relazione. Una tabella per i nomi e una relazione per gli archi. Qua si sta parlando di grafi.



Quando parliamo di **grafi** è fondamentalmente la rete sociale, come Facebook. Su Facebook abbiamo dei nodi che sono le persone e per esempio prendiamo la rete Facebook con tre persone. Gli archi sono i collegamenti (le amicizie nell'esempio). Risulta che A è amico di B, B è amico di C e C ed A non sono amici.



Nel grafico precedente abbiamo una relazione ricorsiva molti a molti, con un attributo. Gli attributi possono essere anche delle relazioni, e ci sono dei pesi che si riferiscono ala coppia di nodi, come nel grafico A e B si conoscono da 7 anni e B e C da solo 1. Se fossimo in Twitter avremmo degli archi orientati in cui A segue B, e B segue C.



Il fatto che A segue B non vuol dire che B segue A, a differenza dell'amicizia che è una **relazione simmetrica**. Se analizziamo le relazioni non simmetriche bisogna assegnare i vincoli.

2.1.1 Application

The nycflights13 dataset contains information about all flights that departed from New York City (e.g. EWR, JFK and LGA) in 2013 and related metadata (planes, airports, airlines and weather conditions). It includes the following entities:

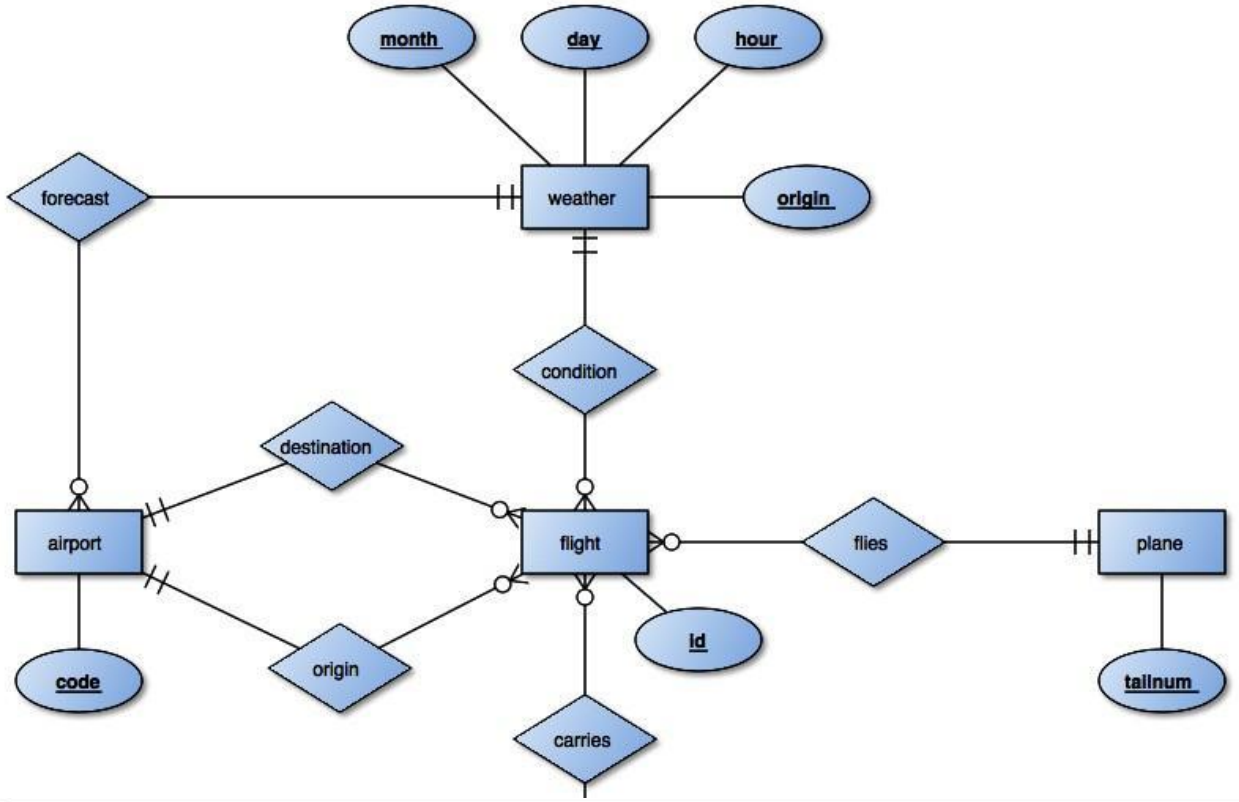
- **flights**: flights departed from NYC in 2013. Attributes include flight id, year, month, day, hour, flight number, origin, destination, actual departure and arrival times, scheduled departure and arrival times, departure and arrival delays, distance. The primary key is flight id.
- **planes**: construction information about each plane. Attributes are: tail number, year manufactured, type of plane, manufacturer, model, number of engines and seats, average cruising speed, type of engine. The primary key is tailnum.
- **airports**: airport names and locations. Attributes are: airport code, name of the airport, location of airport, timezone, altitude. The primary key is aiport code.
- **airlines**: translation between two letter carrier codes and names. The primary key is carrier code.
- **weather**: hourly meteorological data for each airport, including attributes airport, year, month day, hour, temperature, dew-point, humidity, wind direction, speed and gust speed, precipitation, pressure, visibility. The primary key is the combination of airport, year, month day, hour.

The entities are associated with the following relationships:

- a relationship associating a flight with the plane it flew with
- a relationship associating a flight with the airline it flew with
- a relationship associating a flight with the airport of origin
- a relationship associating a flight with the airport of destination
- a relationship associating a flight with the weather conditions of the origin and time of departure
- a relationship associating a weather condition with an airport

Vediamo qua sotto la soluzione a questo problema.

N.B. La relazione tra meteo e aeroporto è invertita




```
knitr::kable(head(nycflights13::planes, 6))
```

tailnum	year	type	manufacturer	model	engines	seats	speed	engine
N10156	2004	Fixed wing multi engine	EMBRAER	EMB-145XR	2	55	NA	Turbo-fan
N102UW	1998	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N103US	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N104UW	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N10575	2002	Fixed wing multi engine	EMBRAER	EMB-145LR	2	55	NA	Turbo-fan
N105UW	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan

2.2 Modello relazionale

Il **progetto logico** di un database consiste nel tradurre il concetto di schema del dato in un schema logico. Il modello logico è indipendente dal modello fisico, cioè la rappresentazione fisica del dato. Il modello logico che noi useremo è chiamato **modello relazionale (RM)**.

Modello relazionale è un modello logico. Quando si progetta un DB si fa un'analisi dei requisiti, poi si fa un modello enti e relazioni, poi lo si trasforma in un modello logico e poi questo modello logico verrà tradotto in un modello fisico. Quello che vedremo noi è il modello relazionale, pensavo da **Codd** in un suo articolo. La sua idea è che il modello sia indipendente dall'implementazione. Cioè io penso al mio modello logico e lo implemento, se cambia l'implementazione allora il modello non cambia. Il modello logico è indipendente dall'implementazione nel caso relazionale.

Il modello relazionale consiste sostanzialmente di due cose:

- **Strutture per organizzare** i dati, tabelle o relazioni
- **Vincoli di integrità**.

Questo modello è basato sul concetto di **relazione** (da non confondere con relazione del modello ER), con questo termine si riferisce sostanzialmente ad una **tabella** (o dataframe in gergo di R). Perché quando costruisco una tabella metto in relazione le varie variabili. Possiamo vedere come dei predicati in logica o come predicati in algebra. Il concetto di relazione qui è formale e deriva dalla teoria degli insiemi.

Il concetto di relazione o tabella può essere capito da chiunque e quindi risulta molto semplice, ma d'altronde è un concetto anche profondo secondo un aspetto geometrico o algebrico. Questo modello ha proprietà teoretiche molto interessanti e molto popolari, e quindi comprensibile da tutti. Vedremo una trattazione poco formale, se si vuole un approccio formale si vada sul link http://users.dimi.uniud.it/~massimo.franceschet/teatro-sql/Kanellakis.pdf.

Consideriamo la relazione Planes, come nel grafico sotto. Abbiamo le **colonne** che sono le **variabili** o **attributi** e le **righe** sono le diverse **unità**.

Ci sono alcune caratteristiche da presentare di una relazione:

- Ci sono le **colonne** e le **righe**
- Le colonne sono degli **attribuiti** con un nome univoco per ogni colonna
- Le righe sono chiamate anche **tuple** e contengono i valori per ogni attributo della tabella
- Una **cella** è l'intersezione tra una riga e una colonna
- L'ordine delle righe e delle colonne non è importante, quindi le righe e le colonne possono essere interpretati come **insiemi**
- Assumendo le colonne come delle variabili, ad ogni di questa è associato un **dominio** (da non confondere con relazione del modello ER), cioè le possibili realizzazioni della variabile (supporto). Possiamo avere domini diversi per ogni tipo di dato. Il dominio è atomico cioè indivisibile.
- Non si possono inserire righe duplicate, questo vuol dire che ci deve essere sempre una chiave
- Come detto prima esiste un **chiave primaria**, che nel caso peggiore sono tutti gli attributi. La chiave non è essenzialmente unica. Il progettista sceglie la chiave primaria. La chiave è una collezione di attributi. Nella pratica spesso ad ogni riga è associato un numero progressivo spesso indicato con id, a volte si usa questo come chiave.
- Gli attributi possono avere **valori nulli**, cioè non presenti: manca il dato. Ci possono esser allora due motivi: il dato non è reperibile, cioè non esiste, o esiste ma non ce l'ho (non ho la mail del candidato). Si ricordi che NULL è costante nulla nei DB mentre NA è la costante nulla in R.

Qui abbiamo la differenza tra schema della relazione e istanza della relazione:

- Lo **schema della relazione** R(X) è composto da il nome R della relazione e un insieme X di attributi. Ad esempio planes(tailnum, year, type, manufacturer, model, engines, seats, speed, engine) e si sottolinea la chiave.
- Una **istanza** invece è un insieme delle righe della relazione. L'istanza parla dei dati, perché essa stessa è i dati, lo schema non tratta dei dati.

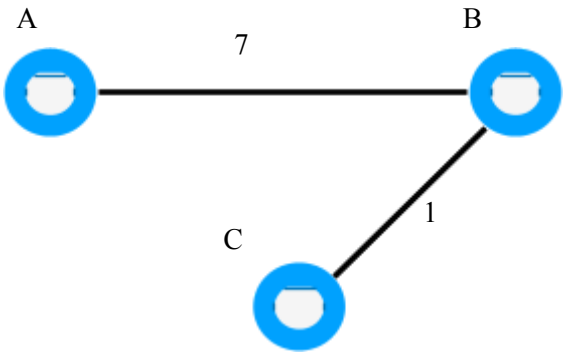
Tipicamente ci sono più tabelle che vengono combinate con una operazione **join**, esse parlano tramite il meccanismo delle chiavi e delle chiavi esterne. Quello che si fa per cui non è mettere tutto in unica tabella, ma in più tabelle come nella practice precedente, una per ogni entità concettuale, e poi si cercherà un modo per collegarle. Collettivamente, tabelle multiple di dati relazionati sono chiamati **database relazionale**. Sia le relazioni che le entità verranno trasformate in tabelle, quindi avremo solo tabelle.

Perciò definiamo:

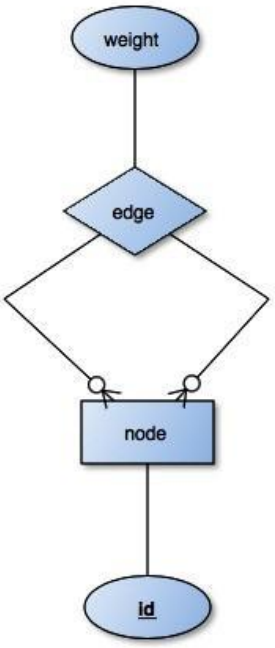
- Un **database schema** come un insieme di schemi di relazioni con nomi differenti
- Una **istanza di un database** (o semplicemente database) è un insieme di istanze di relazioni su un database schema.

Facciamo una osservazione fondamentale: nel modello relazionale, tutta l'informazione è rappresentata tramite una relazione che corrisponde a una tabella con gli attributi. Così, in questo modello, entrambe le entità e le relazioni dello schema concettuale sono implementate nelle relazioni. In particolare, la relazione concettuale tra entità è implementata ad un livello logico tramite le tabelle che contengono valori comuni ad altre tabelle alle quali si riferiscono.

Per esempio prendiamo la rete di un grafo pesato.



In particolare prendiamo uno diretto come segue:



Ipotizziamo che vogliamo tradurre in tabelle. L'idea è creare due tabelle:

- Una per i nodi
- Una per gli archi

Quindi avremo node(id, name) e edge(from,to,weight). Quindi si va a codificare la relazione concettuale mediante dei valori. From è una **chiave straniera** (foreign key) perché in realtà è propria di node però vive in edge.

Questo schema concettuale possiamo trasformarlo in schema logico. Questo sottostante è un esempio di un nodo contenente ciclo di lunghezza 6.

```
node = data.frame(id = 1:6, name = letters[1:6])
edge = data.frame(from = 1:6, to = c(2:6, 1), weight = c(1,1,3,4,2,6))
print(node)
```

```
##   id name
##1  1    a
##2  2    b
##3  3    c
##4  4    d
```

```
##5 5   e
##6 6   f
```

```
print(edge)
```

```
##  from to weight
##1    1  2      1
##2    2  3      1
##3    3  4      3
##4    4  5      4
##5    5  6      2
##6    6  1      6
```

From e to si riferiscono al nodo, per questo sono chiavi straniere, perché si riferiscono ai nodi, ma fanno parte dello schema logico edge. La relazione tra node e edge sono legate per cui a istanze che si ripetono nelle due tabelle.

Il modello relazionale è formato da relazioni e vincoli di integrità:

- **vincolo di dominio**, (es. variabile di tipo intero),
- **vincolo di chiave**, e
- **vincolo di chiave esterna**: ogni valore non nullo della chiave esterna deve corrispondere ad un valore esistente allo schema a cui è riferita.

I vincoli si scrivono con delle frecce in questo modo:
Edge(from) —> node(id)
edge(to)—>node(id)

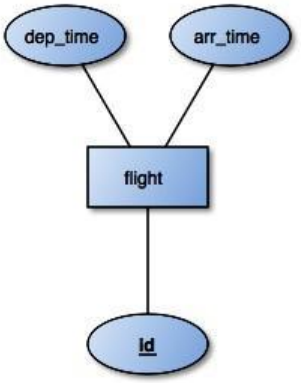
Qui abbiamo ben due nodi, uno da from a id e uno da edge ad id.

2.3 Traduzione da modello ER a quello relazionale

Esiste ora una traduzione da ER a RM, il quale è un algoritmo , che ci traduce uno schema entità relazioni ad un modello relazionale. Le regole sono molto semplici, vediamole una per volta:

2.3.1 Entità

Una entità sarà una tabella con medesimo attributo dell'entità di partenza.



flight(id, dep_time, arr_time)

2.3.2 Relazioni

La cosa più delicata però è la traduzione delle relazioni, ricordando che una relazione è binaria tra due entità con dei vincoli di cardinalità e alle volte questa relazione viene inglobata nelle tabelle delle entità, altre volte non è possibile, per cui sarà necessario creare una tabella per la relazione. La causa di questo problema è la violazione delle chiave primaria.

RELAZIONI UNO A UNO

Riprendiamo la relazione re e regno, in questo caso non occorre prendere una nuova tabella. La relazione possibile è scritta così:

king(name, country)
kingdom(country)
king(country)— > kingdom(country)

kink(name) kingdom(country,
king) kingdom(king)—>
king(name)

In nessuna di queste due possibilità si va contro il vincolo della chiave primaria. Se proprio si vuole essere perfetti è meglio la prima soluzione, perché qualche volta un kingdom non ha un re, quindi il legame tra kingdom(king)—> king(name) potrebbe dare un valore nullo. In realtà si dovrebbe fare un altro ragionamento: la scelta dipende anche da come viene usato il DB, cioè dalle interrogazioni tipiche che mi aspetto che verranno utilizzate di più. Se nel mio DB so che tutti gli utenti sono interessati a cercare le nazioni su cui un re governa è meglio la prima, basta usare la prima tabella in cui abbiamo re e country, nella seconda casistica dovremo fare più passaggi.

RELAZIONE UNO A MOLTI

In questo caso non c'è ambiguità e c'è una sola soluzione. Prendiamo la relazione tra flight e planes. Allora la traduzione sarà la seguente:

flight(id,trailnum)
plane(tailnum)
flight(tailnum) —> plane(tailnum)

Se introducessi flight in plane, cioè plane(tailnum, flight) e flight(id), succede che siccome so che ad ogni aeroplano possono essere associati più voli perderei il vincolo di integrità della chiave primaria. Se abbiamo un aeroplano che conduce due voli avremo due righe duplicate. Conviene per cui buttare la relazione nella tabella dell'entità che ha come riferimento massimo una entità, in questo caso flight.

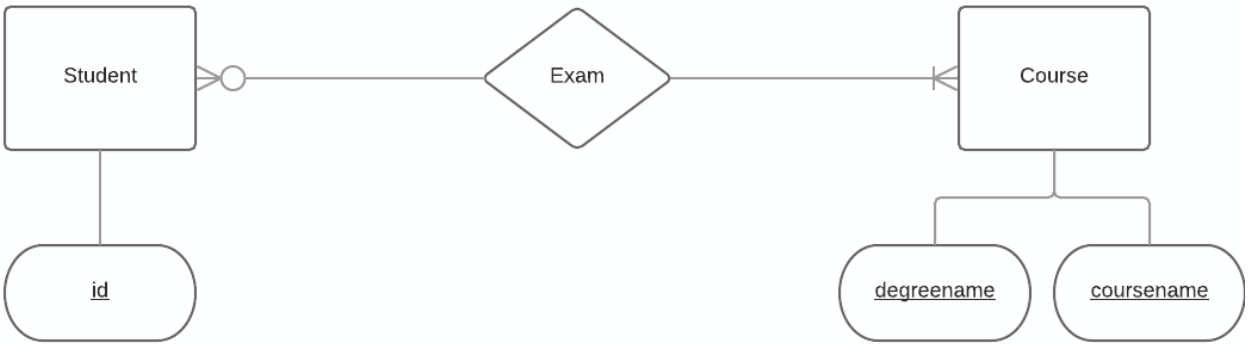
RELAZIONE MOLTI A MOLTI

Prendiamo la relazione tra studente e corso. In questo caso ovviamente non posso buttare la relazione ne da una parte ne dall'altra.

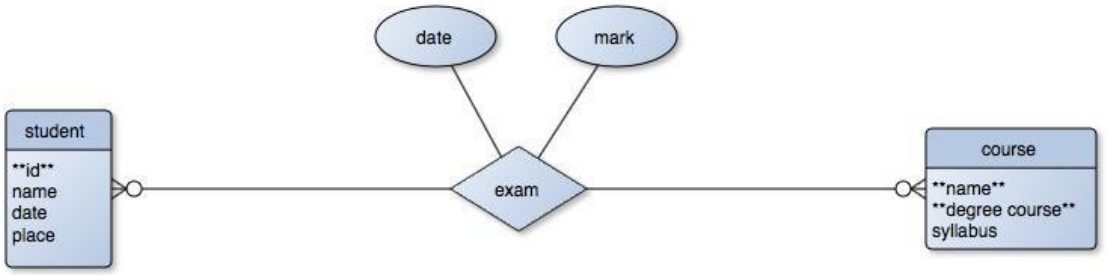
student(code)
course(nome)
enroll(student, course)
enrol(student)—>student(code)
enroll(course)—>corse(name)

Applicando queste quattro regole dovremmo essere in grado di fare delle traduzioni.

For each of the following universes of discourse:

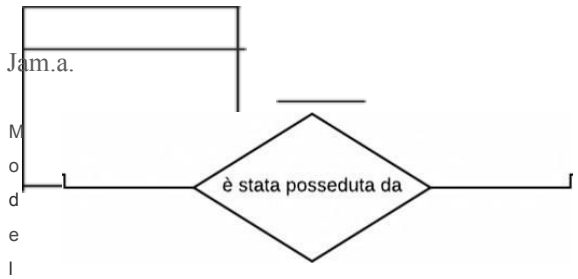


student(id, name, birth date, birth place)
course(name, degreename, syllabus)
exam(studentid, coursename, degreename, date, mark)
exam(studentid) ---> student(id)
exam(coursename, degreename) ---> course(name, degreename)



student(id, name, date, place)
course(name, degree_course, syllabus)
exam(student, course, degree_course, date, mark)
exam(student) –> student(id)
exam(course, degree_course) –> course(mame, degree_course)

CAR



Jam.a.

Modello Macchina Colore

Macchina
Guidatore

Codice Eiscale

Nome

Cognome

macchina(targa modello, marca, colore)
guidatore(l;;.F. Nome, Cognome)
proprieta(targa ~ dataInizioPossesso.
dataFinePossesso
proprietaitarga)--> macchina(targa)
proprieta(CF) --> guidatore(CF)

date

person

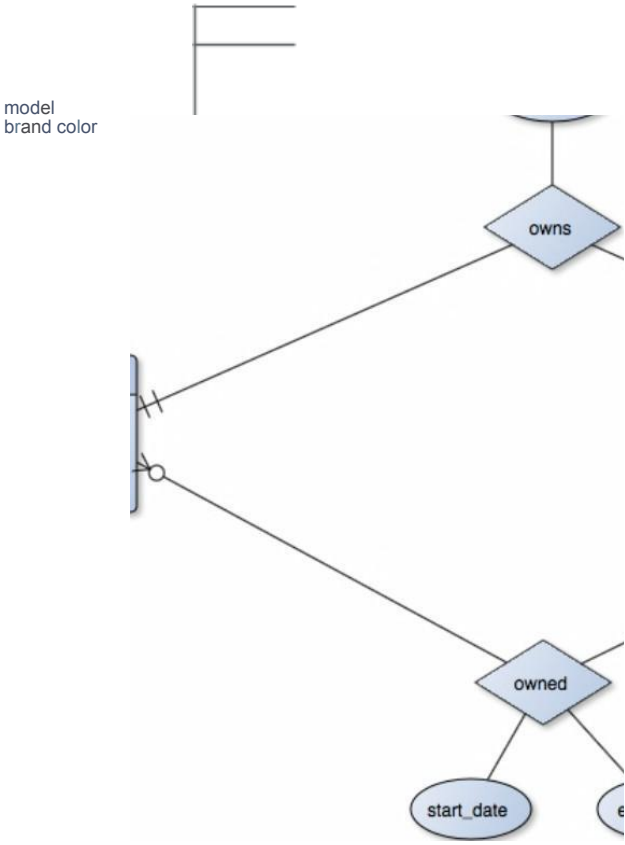
name
surname

car

.. llcenoe_plate..

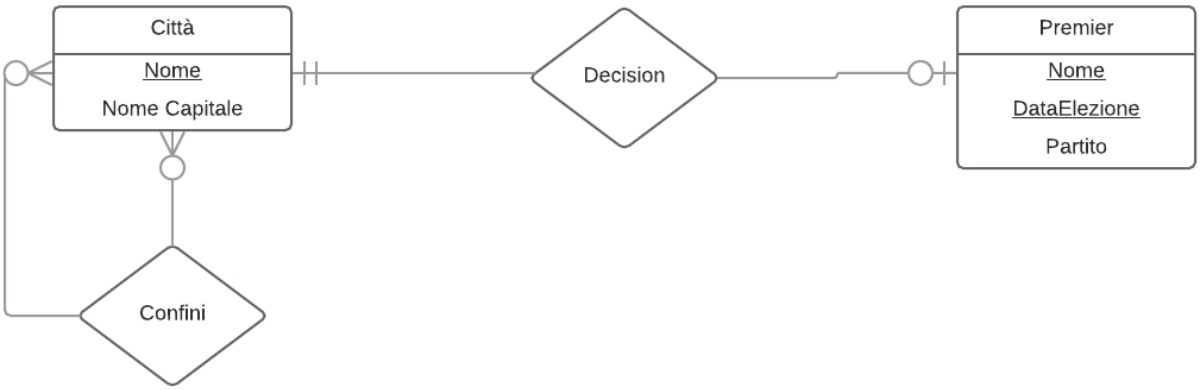
person(tax_code, name, surname)
carQicense plate, model, brand, color, owner, date)
owned(person, car, starting_date, ending_date)
car(owner) -> person(tax_code)

owned(person)->
person(tax_code)
owned(car) -> car(license
plate)



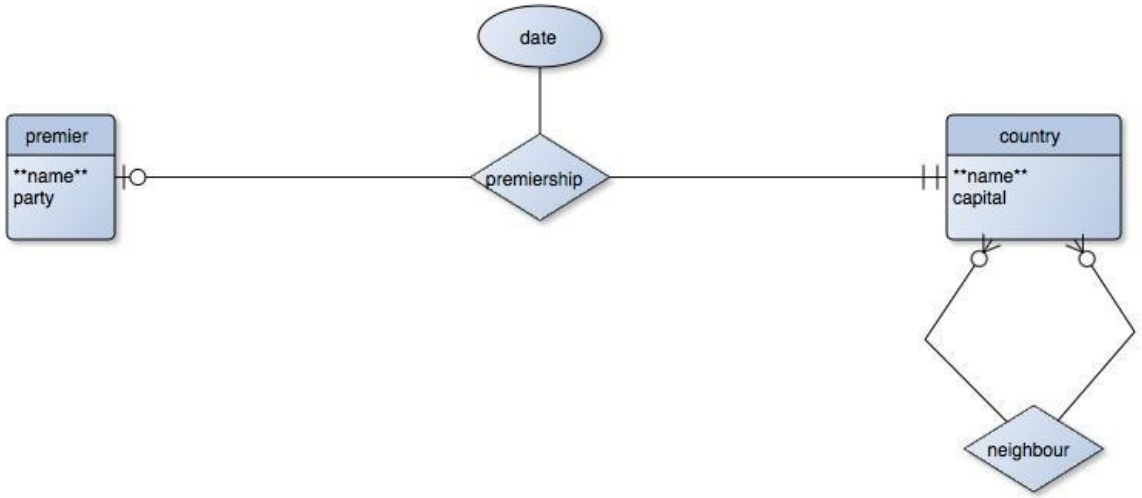
H
tab

co



Città(Nome, NomeCapitale)
Premier(Nome, DataElezione, Partito, NomePaese)
Premier(NomePaese)-->Città(Nome)

Confini(NomePaese1, NomeConfine2)
Confini(NomePaese1)--> Paese(nome)
Confini(NomePaese2)--> Paese(nome)

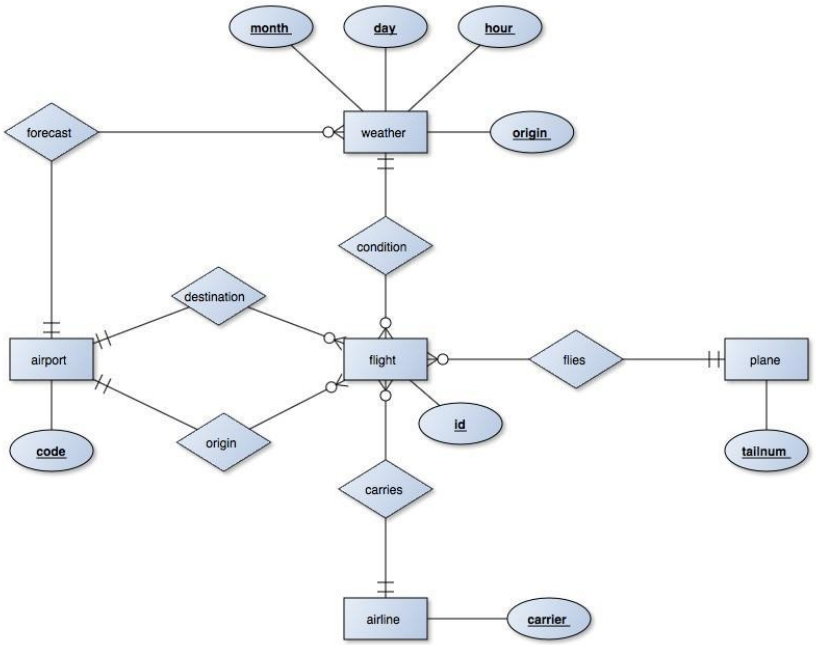


> country(name, capital, premier, election_date)
> premier(name, party)
> neighbour(country1, country2)
> country(premier) –> premier(name)
> neighbour(country1) –> country(name)
> neighbour(country2) –> country(name)

La relazione si basa sullo stato attuale

Practice 2

Translate the ER schema below for the NYC flight dataset into a relational model. For simplicity, make the translation for key attributes only.




```
plane(tailnum)
airline(carrier)
airport(code)
weather(hour, day, month, airport_origin_code)
flight(id, plane_tailnum, airline_carrier, airport_origin_code,
airport_arrival_code, partenza_ora, partenza_giorno, partenza_mese, arrivo_ora,
arrivo_giorno, arrivo_mese)

flight(plane_tailnum) -> plane(tailnum)
flight(airline_carrier) -> airline(carrier)
flight(airport_origin, partenza_ora, partenza_giorno, partenza_mese) ->
weather(aiport_origin_code, hour, day, month)

weather(airport_origin_code)->airport(code)
```

MANCANO LE SOLUZIONI CHE DOVREBBE CARICARE

2.4 SQL

SQLite è un **DBMS**, molto usato in data science. È un linguaggio di interrogazione e manipolazione e definizione dei dati. Questa è la parte della progettazione fisica, cioè la progettazione in cui andiamo di fatto a creare le tabelle, a popolarle e ad interrogarle. Tutto questo viene fatto con un unico linguaggio universale chiamato **SQL** ed è un linguaggio che ci permette la definizione dei dati, cioè creare gli schemi di relazione e i vincoli di integrità, ma anche è un linguaggio di manipolazione dei dati (inserire, cancellare, modificare i dati) e ci consente di recuperare dati tramite la query.

Questo linguaggio fu proposto dall'articolo di Codd sopracitato ed è di fatto lo standard da 50 anni per interrogare le basi di dati relazionali. Uno scienziato del dato è bene che sappia un po' di SQL per vari motivi, anche se magari non viene usato. Il motivo più ovvio è che i dati non stanno in memoria e serve SQL per estrarli per poi essere elaborati in R o in Python.

Esso è un **linguaggio dichiarativo** e non imperativo, perché dichiaro solo quello che voglio e la macchina traduce la mia dichiarazione in un piano di estrapolazione di questa richiesta. Il linguaggio imperativo è un linguaggio che fa ciò che diciamo, diamo dei comandi. Si dichiarerà solo il nostro intento, si specifica cosa

vogliamo recuperare ma non dicamo come recuperare. Quando una query viene eseguita un programma detto **ottimizzatore** trasforma la query in un **piano di accesso ai dati**, una sorta di algoritmo che accede ai dati in modo efficiente. Qua non occorre saper programmare quindi, ma basta saper esprimere il nostro "bisogno" in modo corretto.

Faremo una lunga DEMO in cui introdurremo tutte le caratteristiche di SQL. Si userà SQLite sul data set New York Flights. SQLite ha le seguenti caratteristiche

- È **transazionale**, cioè gestisce le transazioni
- **Zero-configuration**, non occorre esser esperti per utilizzare e configurare SQLite, a differenza degli usuali DBMS.
- Ogni DB creato è salvato in un **unico file**, pur avendo mille tabelle, poi si potrà semplicemente zippare ed inviare a qualche amico
- Ha un codice molto piccolo di dimensione
- Ha un'interfaccia di programmazione molto facile, si può usare anche da altre applicazioni, come faremo noi con R.
- È **veloce**, alcune volte più veloce all'accesso diretto
- È scritto in **linguaggio C**
- Disponibile in un unico file C
- È **autocomprensivo**, non ha dipendenza esterne
- Può essere utilizzato in tutte le piattaforme
- Ha un'interfaccia non troppo intuitiva che inizialmente useremo.

Tra i consigli di utilizzo di questo SQLite c'è anche l'analisi dei dati o meglio si presta bene per archiviare i dati che poi verranno analizzati in altri linguaggi come R. Quando abbiamo dei dataset grandi possiamo archiviare i dati in un database, ad esempio in SQLite, e poi farli a fette, selezionando le fette che ci servono per poi utilizzarle ed analizzarle.

Un altro caso d'uso è l'utilizzo come database durante un corso, perché è facile da installare, scaricare e compilare. Quindi molto utilizzato dai professori per accedere a questo argomento dei DB. Per esempio per fare il progetto alla fine dell'anno si potrà mandare anche il DB, il quale sarà semplicemente un file che si può immaginare come un grande .csv, quindi con un gran numero di dati. Si guardi il sito se ci si vuole sbizzarrire. Noi abbiamo interesse verso questi due casi d'uso sopra illustrati, ma ce ne sono illustrati di altri sul sito.

Le **transazioni** sono sequenze di comandi che ci permetto di mantenere consistente il nostro DB. Alla fine della sequenza di questi comandi possiamo fare l'istruzione di **commit** che rende effettive le richieste oppure possiamo tornare indietro, con l'istruzione ***roll back*** (disfa tutto). Quindi alla fine di una transazione il DB rimane consistente. Se per esempio faccio un bonifico, ci sono due operazione addebitare e accreditare. Queste sono due azioni da inserire in una transazione. Quindi inserendo queste due operazioni possiamo creare una transazione e possiamo accettare la transazione o tornare indietro, se le operazioni sono o meno venute con successo. Quindi SQLite supporta queste transazioni. Può in più sopportare dati relativamente grandi, circa 140 TeraByte.

Una cosa che si suggerisce è quella di inserire la cartella scaricata nel *path* di sistema, in modo che si possa richiamare da qualsiasi altro fonder il DB. Lavoreremo con new york flights.

2.4.1 Data Definition e Data Management

Vediamo ora la demo tramite dei passi che ci portano a costruire una base di dati tramite una DB. Questo perché alcune volte la memoria centrale non è sufficiente a tenere l'intero DataSet, e quindi un buon metodo è quello di creare un buon DB e quando ci servono fette di dati li estraiamo con SQL e li utilizziamo per fare delle nostre analisi. Se il nostro DataSet contiene un numero di dati che può essere caricato in memoria centrale, allora lo si usi senza sto ambaradan di SQL.

1. Scaricare un DBMS, per noi SQLite
2. Vedere che il DataSet sia un minimo consistente. In Data Science i vincoli di integrità sono meno evidenti rispetto a quello che abbiamo visto fino ad ora. Certo sarebbe meglio se i dati fossero verificati per avere dei dati consistenti. Come compromesso a noi bastano dei DataSet con delle chiavi, cioè righe non duplicate.
3. Esportare i file in formato cdv
4. Creare il DB
5. Creare le tabelle, con i vincoli di integrità

6. Inserire i dati

7. Creare gli indici
8. Fare un backup
9. Provare a violare i vincoli
10. Fare un restore del DB dal backup

SCARICARE E COMPILARE SQLITE

La prima cosa è creare una cartella di lavoro e poi con r studio si creare un progetto (file, new project). Si sceglie la cartella ed aprendo il terminale con il comando change directory "cd" si sceglie la cartella dove abbiamo messo il progetto di R. Con il comando ls sul terminale si vedono tutti i contenuti della cartella. Per uscire si faccia il comando .quit, se facciamo $PATH si vede il contenuto della cartella PATH.

AGGIUNGERE CHIAVI ESTERNE

Ora per aggiungere le chiavi surrogate, dall'ambiente di R Studio, dobbiamo fare questi comandi:
In primo luogo contiamo quante sono le chiavi che si presentano più di una volta.

```
library(nycflights13)
library(dplyr)

flights %>%
  count(year,  month, day, sched_dep_time, flight, carrier) %>%
  filter(n >  1)
```

Essendo il risultato zero, allora la chiave rende le tuple uniche ed essendo la chiave troppo lunga aggiungiamo una chiave surrogata con il comando mutate e lo riporto all'inizio della tabella:

```
flights  <- flights
  %>%
  arrange(year, month, day,  sched_dep_time) %>%
  mutate(id =  row_number()) %>%
  select(id,  everything())
```

Creata la chiave surrogata, possiamo verificare che le altre chiavi siano effettivamente chiavi, questo è molto importante perché i DB vogliono che almeno i vincoli di chiavi primarie siano soddisfatte, se non lo sono bisogna aggiungere delle chiavi surrogate:

```
planes %>% count(tailnum)
  %>% filter(n >  1)

airports %>%
  count(faa) %>%
  filter(n >  1)

airlines %>% count(carrier)
  %>% filter(n >  1)

weather %>%
  count(year, month, day, hour, origin)  %>%
  filter(n >  1)
```

ESPORTARE I DATAFRAMES IN FILE CSV

A questo punto generiamo i file csv dalla libreria, perché il mio obiettivo è importare il mio DataSet in una base di dati:

```
write.table(airlines, file =  "airlines.csv",  sep =  ",",  row.names=FALSE,  col.names  = FALSE, quote=FALSE, na="",
qmethod =  "double")
```

```
write.table(airports, file = "airports.csv", sep = ",", row.names=FALSE, col.names = FALSE, quote=FALSE, na="", qmethod = "double")

write.table(planes, file = "planes.csv", sep = ",", row.names=FALSE, col.names = FALSE, quote=FALSE, na="", qmethod = "double")

write.table(weather, file = "weather.csv", sep = ",", row.names=FALSE, col.names = FALSE, quote=FALSE, na="", qmethod = "double")

write.table(flights, file = "flights.csv", sep = ",", row.names=FALSE, col.names = FALSE, quote=FALSE, na="", qmethod = "double")
```

Comprendiamo meglio il comando:

```
write.table(airlines, file = "nomefile.csv", sep = ",", row.names=FALSE (non voglio il nome delle righe), col.names = FALSE (non voglio il nome delle colonne), quote=FALSE (non voglio che le stringhe non abbiamo le virgolette), na="" (per scrivere tutti i valori na come stringa vuota), qmethod = "double" )
```

qmethod riguarda la problematica della doppia presenza delle doppia virgoletta "mamma dice "ciao"", e quindi serve per definire come sorpassare questo problema. A questo punto nella nostra tabella abbiamo i file csv che conterranno i nostri dati.

CREARE UN DATABASE IN SQLITE

A queso punto si va sul terminal e

```
cli235-54:~ andreapesce$ cd /Users/andreapesce/Desktop
cli235-54:Desktop andreapesce$ cd SQLite
cli235-54:SQLite andreapesce$ SQLite3 nycflights13
```

Quello scritto in neretto è quello che effettivamente si ha scritto sul terminale. A questo punto creiamo delle cartelle. Prima di tutto bisogna scaricare il file e metterlo nella cartella. Una volta messo su R Studio basterà fare file e aprire il file appena scaricato. Con il comando .databases si possono vedere i database nella cartella in cui ci siamo posizionati con il comando cd. Conviene ogni volta che si crea un nuovo di R con new project.

CREARE TABELLE CON CHIAVI ESTERNE E PRIMARIE

INSERIRE DATI NELLE TABELLE USANDO I FILE CSV ESPORTATI

A questo punto inseriamo i dati, cioè creiamo uno schema, la prima cosa da fare è utilizzare un separatore adeguato. Si scarichi il file create dal sito e lo si metta nel working directory. Sul terminal, basta ora scrivere

```
.separator ,
.mode csv
```

Prima di tutto dobbiamo creare le tabelle, vediamo con i comandi qua sotto che dobbiamo dichiarare il dominio di ogni campo e definire qual è la chiave primaria e quali sono le chiavi esterne:

```
CREATE TABLE airlines (
 carrier VARCHAR(2),
 name VARCHAR(60),
 primary key (carrier)
);

CREATE TABLE airports (
```

```
faa VARCHAR(3),
name VARCHAR(60),
lat REAL,
```

```
 lon REAL,
 alt INTEGER,
 tz INTEGER,
 dst VARCHAR(1),
 tzone VARCHAR(60),
 primary key (faa)
);

CREATE TABLE planes (
 tailnum  VARCHAR(10),
 year INTEGER,
 type VARCHAR(30),
 manufacturer VARCHAR(30),
 model VARCHAR(30),
 engines INTEGER,
 seats INTEGER,
 speed INTEGER,
 engine VARCHAR(30),
 primary key (tailnum)
);


CREATE TABLE weather (
 origin VARCHAR(3),
 year INTEGER,
 month INTEGER,
 day INTEGER,
 hour INTEGER,
 temp REAL,
 dewp REAL, humid
 REAL, wind_dir
 REAL, wind_speed
 REAL, wind_gust
 REAL, precip
 REAL, pressure
 REAL, visib
 REAL, time_hour
 REAL,
 primary key (year, month, day, hour, origin)
);

CREATE TABLE flights (
 id BIGINT, -- big integer
 year INTEGER,
 month INTEGER,
 day INTEGER,
 dep_time INTEGER,
 sched_dep_time INTEGER,
 dep_delay INTEGER,
 arr_time INTEGER,
 sched_arr_time INTEGER,
 arr_delay INTEGER,
 carrier VARCHAR(2),
 flight INTEGER,
 tailnum VARCHAR(10),
 origin VARCHAR(3),
 dest VARCHAR(3),
 air_time INTEGER,
 distance INTEGER,
 hour INTEGER,
```

```
 minute INTEGER,
 time_hour REAL,
 primary key (id),
 foreign key (carrier) references airlines(carrier),
 foreign key (tailnum) references planes(tailnum),
 foreign key (origin) references airports(faa),
 foreign key (dest) references airports(faa),
 foreign key (year, month, day, hour, origin) references weather(year, month,
day, hour, origin)
);
```

Sul sito c'è il link che porta al sito di SQLite che porta alla spiegazione formale di tale comando. Altri comandi interessanti sono *drop table* o *alter table*, per modificare lo schema, non il contenuto. A questo punto siamo pronti per la grande importazione e basterà immettere questi comandi:

```
.import airlines.csv airlines
.import airports.csv airports
.import planes.csv planes
.import weather.csv weather
.import flights.csv flights
```

Con questi prossimi invece vediamo le prime dieci righe delle tabelle:

```
select * from airlines limit 10;
select * from airports limit 10;
select * from planes limit 10;
select * from weather limit 10;
select * from flights limit 10;
```

Con il comando .modecolum si riesce a vedere i dati richiesti in maniera più adeguata. Ora vogliamo che NA corrisponda a NULL

```
update flights set dep_delay = NULL where dep_delay = "";
update flights set arr_delay = NULL where arr_delay = "";
update flights set dep_time = NULL where dep_time = "";
update flights set arr_time = NULL where arr_time = "";
```

E a questo punto verifichiamo che alcuni vincoli non sono rispettati con il comando anti_join su R. Ci serve per vedere se ci sono dei voli, ad esempio, che hanno un tailnum che non è presente nella tabella planes.

```
flights %>%
 anti_join(planes, by = "tailnum") %>%
 count(tailnum, sort = TRUE)
```

Facendo similmente si può vedere la presenza di questo errore in destination.

```
flights %>%
 anti_join(airports, by = c("dest" = "faa")) %>%
 count(dest, sort = TRUE)


flights %>%
 anti_join(weather, by = c("year" = "year", "month" = "month", "day" = "day", "hour" = "hour", "origin" = "origin"))
%>%
 count(year, month, day, hour, origin, sort = TRUE)
```

A questo punto cosa potremmo fare??? L'idea più stupida potrebbe essere eliminare le righe che non combaciano, ma così perdo informazioni. Se no, la scelta opposta, è lasciare così. Una soluzione intermedia,

invece di cancellare un'intera riga di flights si mette a NULL semplicemente l'attributo tailnum, si ricorda che i vincoli di integrità referenziale viene valutata solo due dati non NULL, quindi non ci darà più "errore". Oppure potrei aggiungere il tailnum su Planes e fare un po' più di lavoro cercando le informazioni mancanti.

Forse la cosa più facile e corretta è quella di mettere NULL. In questo caso non abbiamo fatto nulla, notando che il DB non ha protestato fino ad ora, perché di default le verifiche delle chiavi esterne devo attivarle io, mentre quel di chiavi primaria sono attivate di default.

Quindi una volta in possesso del DB bisogna verificare i vincoli di integrità, se le chiavi primaria non è rispettato allora eliminiamo i duplicati.

CREARE INDICI

Ci potrebbe esser utile a questo punto creare degli **indici**, molto utili per aumentare l'efficienza delle query. Si possono creare sia a tabelle vuote che con i dati. Se creiamo gli indici prima di mettere i dati il costo è zero, però se già li ho creati e poi metto i dati il costo è maggiore perché si aggiorna ad ogni dato immesso. Se creo l'indice alla fine il tutto costa di meno. Un indice è: un computer ha due tipi i memoria, una è quella primaria o centrale, ed è veloce con accesso diretto e non sequenziale e in più è volatile, cioè viene azzerata quando il computer si spegne. L'altra è una memoria secondaria, quello che ora è una memoria poco costosa però lenta, perché il disco gira ma la puntina deve beccare la traccia e spedirla in memoria centrale. Il computer per elaborare l'informazione deve portare i dati in memoria centrale non li può elaborare dalla memoria secondaria. Essendo la primaria costosa ce n'è solitamente di meno. Il processore per elaborare informazioni ha necessità di portare i dati dalla memoria secondaria a quella primaria. I DB sono solitamente situati nella memoria secondaria e vengono caricate solo le informazioni che ci servono in quell'istante. Quindi il vero collo di bottiglia dei DB è la velocità degli accessi alle informazioni nella memoria secondaria tramite la primaria. Cioè un buon programmatore cerca di minimizzare gli accessi al disco. Un modo per minimizzare ciò sono gli indici. Come ad esempio gli indici del libro. È più efficiente perché un indice è **piccolo** e perché è una **struttura ordinata**. Sappiamo che accedere ad una struttura i dati ordinata è più efficiente che accedere ad una che è disordinata, dove l'unico modo per fare una ricerca è con metodo sequenziale (in un libro sarebbe ricercare pagina per pagina l'informazione). Quindi gli indici non sono nient'altro che una struttura di dati ordinata che mi concede di accedere al dato in maniera più efficiente. Una buona pratica è quella di creare degli indici sulle chiavi primarie (solitamente fatto in automatico) e quelle esterne. Un'altra buona pratica è creare un indice su attributi molto utilizzati. Attenzione che mantenere un indice è costoso, perché se aggiorno la tabella devo aggiornare anche gli indici. Quindi se devo aggiornare una tabella più volte conviene mettere indici solo sulle chiavi, se no l'aggiornamento diventa molto lento. Creiamo degli indici sulle chiavi primarie ed esterne:

```
create unique index airlines_index on airlines(carrier);
create unique index airports_index on airports(faa);
create unique index flights_index on flights(id);
create unique index planes_index on planes(tailnum);
create unique index weather_index on weather(year, month, day, hour, origin);
```

Auspicando che questi indici ci siano utili.

```
create index flights_tailnum on flights(tailnum);
create index flights_origin on flights(origin);
create index flights_dest on flights(dest);
create index flights_carrier on flights(carrier);
create index flights_weather on flights(year, month, day, hour, origin);
```

BACKUP DEL DB

A questo punto facciamo il backup.

```
.backup nycflights13_backup_20180313
```

A questo punto si possono distruggere i i database tanto abbiamo salvato una copia.

VIOLARE I VINCOLI DI INTEGRITÀ

Potremmo ora per esempio far incazzare il nostro DB andando ad affliggere i vincoli di integrità.
Inseriamo un dato nella tabella con il comando insert:

```
insert into airlines values ("9E", "My Air");
```

Per fortuna il comando ci da un errore, perché in questo modo rifacendo il comando di prima

```
select * from airlines;
```

Non abbiamo una riga ripetuta. Per attivare il controllo dei vincoli delle chiavi straniere:

```
PRAGMA foreign_keys = ON;
```

Questo comando fa un controllo avanti nel tempo e non prima, quindi se da questo punto in poi non si fanno errori. A questo punto non possiamo più violare i vincoli di integrità. Proviamo ad aggiungere un nuovo volo con id 0 e Carrier xy.

```
insert into flights(id, carrier) values ("0", "XY");
```

Anche in questo caso da errore per la violazione del vincolo di chiave esterna. Come notiamo nel comando insert abbiamo solo aggiunto due campi, gli altri vengono messi di default come NULL.
A questo punto proviamo a violare nuovamente i vincoli di integrità referenziale cercando di cancellare con il comando delete dalla tabelle Airlines il Carrier di tipo UA, se la cancelliamo abbiamo uno stato inconsistente.

```
delete from airlines where carrier = "UA";
```

Allo stesso modo se cerchiamo di aggiornare il campo UA con il campo XY

```
update airlines set carrier = "XY" where carrier = "UA";
```

Tutti questi comandi ci danno errore perché ci troveremo in uno stato inconsistente.

RECUPERARE DAL BACKUP

A questo punto se abbiamo fatto dei danni con il comando Restore e grazie al backup possiamo rigenerare il punto da dove abbiamo fatto il backup.

```
.restore nycflights13_backup_20180215
```

E vediamo la comodità della data che ci aiuta vedere che database vogliamo rigenerare.

QUERY

Ora vediamo le query, parte fondante delle interrogazioni. Abbiamo visto come creare gli schemi, l'ultima parte del linguaggio SQL ha a che fare con le interrogazioni vere e proprie.

SELECT E FROM

C'è il comando **select** che risulta essere molto complicato e ha lo scopo di recuperare questi dati in una certa forma, ovviamente non tutto il DB ma solo una parte dei dati. Lo facciamo non con un linguaggio imperativo quindi, ad esempio, siamo interessati ad avere gli studenti che hanno passato "analisi matematica" al primo colpo con voto superiore a 28. Con questi dati poi possiamo usarli per altre analisi magari in R.

```
select *
from flights
```

La sintassi è parecchio complessa, la parole più importi sono from, where, group by, having, order by. Vedremo una a una queste procedure. La prima clausola è **select from**. Select significa seleziona alcuni campi delle tabelle, from mi dice quali tabelle usare. Il simbolo *, utilizzato sopra sta per tutti i campi, in questo caso da flight. Però possiamo usare una clausola **limit**, ad esempio limit 10 avremo le prime 10 righe se metto select e il nome di alcune variabili selezioniamo solo alcuni campi delle tuple, si fa una cosiddetta proiezione, proiezione sarebbe sinonimo di selezione ed è detta proiezione in algebra lineare.

```
select id, flight, carrier
from flights
```

Facendo un file di testo su R con tali comandi e salvandolo basterà fare sul terminal .read NomeFile direttamente incuterà i comandi scritti nel testo. Quindi abbiamo eseguito la prima interrogazione sul nostro DB. Ora ne creeremo di più complicate.

WHERE

Vediamo ora la clausola **where** che filtra le righe della nostra tabella. Molto spesso siamo interessati a selezionare solo alcune righe, nonché quelle che soddisfano un qualche predicato, per esempio possiamo selezionare tutti gli aeroplani che sono costruiti da AIRBUS INDUSTRIE, quindi con la clausola where costruiamo una condizione, potrebbe essere essere semplice come una uguaglianza o più complicata, come nei casi successivi:

```
select *
from planes
where manufacturer = "AIRBUS INDUSTRIE”
limit 10

select *
from planes
where engines > 2
```

Nota: per fare un commento su SQL si faccia il doppio trattino --. Con la prima query troviamo i primi dieci aeroplani costruiti da AIRBUS INDUSTRIE. Con questa clausola where immettiamo un **predicato booleano** sulle tuple. Un predicato è una formula logica che ha un valore di verità VERO o FALSO. Un predicato atomico è costituito da una espressione o meglio ha la forma del tipo: è confronto tra due espressioni $X\ o\ pY$, dove X e Y sono delle costanti o sono delle variabili (nomi colonne) op è un operatore.
Gli operatori possibili sono = , < > ($d\ i\ v\ e\ r\ s\ o$), < , > , < = , > = oppure possiamo usare like e _ che vale per ogni valore o like con % vale per ogni stringa anche vuota. Se per esempio scrivo

NOME like “_MA”

Sono tutti i normi che iniziano per un qualsiasi carattere e poi seguiti da MA, se invece uso

NOME like “%MA”

Sono tutti i nomi che terminano per MA. I caratteri atomici sono questi che possono essere combinati con i connettivi booleani and, or e not. Per verificare se un valore è uguale a NULL si usa il predicato *is null* o *is not null.* Perché se c’è nulla vuol dire che è un dato di cui non sappiamo nulla, magari esso è un valore esistente o inesistente, essendo un caso particolare si usano quindi questi due predicati.

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

Facciamo questo esercizio:

Tutti i voli partiti a natale:

```
select *
from flight
where day=25 and month=12
limit 20;
```

Come vediamo abbiamo una congiunzione per la quale entrambi i congiunti devono essere veri. Tutti i voli che hanno come valore nullo deporture o arrival delay:

```
select *
from flights
where dep_delay is null or arr_delay is null
limit 20;
```

Come vediamo abbiamo una congiunzione per la quale basta che solo uno dei due congiunti sia vero. Tutti i voli che non hanno departure time NULL e non hanno NULL arrival time:

```
select *
from flights
where dep_time is not null and arr_time is not null
limit 10;
```

Voli che hanno la variabile tailnum che inizia con N10:

```
select *
from planes
where tailnum like "N10%"
limit 20;
```

Per ora non abbiano visto nulla di difficile SQL infatti è parecchio intuitivo. Bisogna però far attenzione ai NULL. Se scriviamo

“MAX”=NULL

(nel caso di R sarebbe ==) Questo intanto è un predicato perché abbiamo due costanti e un operatore. Se per null intendiamo che non è disponibile l’informazione allora questo risultato non ha ne valore vero ne falso, perché null nasconde qualsiasi valore, magari anche “MAX”. Quindi la logica in SQL non è binaria ma è a tre valori di verità: true, false e unknown. Un qualsiasi confronto con valore NULL darà come risultato unkonown.

NULL = NULL
Questi potrebbero essere uguali o potrebbero essere diversi, quindi anche in questi casi è unknown. Risulta che in questi casi il risultato è sconosciuto e potrebbe assumere qualsiasi valore. Ovviamente se facessimo is.na(NA) da True e !is.na(NA) da False. Se in più facessimo TRUE & NA da unknown e FALSE & NA da False, perché qualsiasi valore di NULL sarà sempre false. Similmente TRUE or NA è True, Invece FALSE or NA è Unknown.
Se facessimo ora:

```
select *
from planes
Where (engines>2) or (engines <=2)
```

Se ci fossero dei valori nulli, confrontando valori nulli con una costante avremmo come risposta nullo, e volendo come risultato quelli con risultato TRUE, e quindi si seleziona solo gli aeroplani che hanno valore non nullo. Se per assurdo tutte le tuple avessero in questo campo NULL allora restituirebbe zero aeroplani

ORDER BY

Vediamo ora **order by** che serve per ordinare le tuple rispetto a qualche campo, per esempio se facessimo:

```
select *
from planes
order by engines
```

Avremo tutti gli aerei in ordine crescente di numero di motori. Per fare in ordine decrescente:

```
select *
from planes
order by engines desc
```

Quindi dal più alto al più basso. Se vogliamo ordinare per più campi, magari per nome e poi per cognome, basta scrivere la lista dei campi come *order by nome, cognome*.
Proviamo ora a fare i voli che hanno ritardo positivo ordinati in ordine decrescenti per ritardo:

```
select *
from flight
where dep_delay>0
order by dep_delay desc
limit 50;
```

Selezioniamo tutti i voli che hanno recuperato in volo rispetto ai ritardi.

```
select *, (arr_delay-dep_delay) as catchup
from flights
where catchup < 0
order by catchup desc
limit 20;
```

Dove (arr_delay-dep_delay) as catchup creiamo una nuova variabile per l'interrogazione.

GROUP BY E HAVING

Invece **group by** e **having** sono molto utilizzati in statistica nella quale si cercano dei gruppi, magari per sommare qualche valore di alcune variabili, come creare i gruppi maschi e femmine e fare una media dei voti. Innanzitutto abbiamo degli **operatori aggregati** che calcolano una statistica riassuntiva su un insieme di tuple come sommare, massimo, minimo, media, contare. I classici operatori sono: **count**, **min**, **max**, **avg** (avarage), **sum**. Quindi se vogliamo questi operatori aggregati si possono utilizzare su tutta la tabella o su gruppi. Se volgiamo calcolare il numero di tuple si può fare:

```
select count(*)
from planes
```

Per contare invece il numero di manufacturer non nulli basterà scrivere:

```
select count(manufacturer)
from planes
```

Se voglio contare i numeri distinti di manufacturer basterà scrivere:

```
select count(distinct manufacturer)
from planes
```

Applicando questi comandi essendo i due primi risultati uguali nessun manufacturer risulta essere nullo. In ultimis si riassume le principali statistiche:

```
select max(seats), min(seats), sum(seats), avg(seats), sum(seats) / count(seats)
from planes
```

Se ci sono valori nulli la media viene fatta sui valori non nulli a differenza di R in cui si vuole che si dica se si vuole fare con o senza i valori nulli.

Come si diceva si usano questi operatori aggregati su gruppi, per creare dei gruppi basta usare la formula group by e farli seguire dal campo. Se raggruppassi per sesso avrei gruppo con sesso M e gruppo con sesso F, se raggruppassi per età avremmo i gruppi per ogni età. Si possono raggruppare per due attributi, solitamente non si fa mai per più di due. Su ognuno di questi gruppi poi applico la mia statistica.

```
select dest, count(*) as count
from flights
group by dest
```

Questo comando raggruppa per destinazione il numero di voli. Vogliamo ora ordinare per destinazione i voli:

```
select dest, count(*) as count
from flights
group by dest
order by count desc
```

Vediamo ora quali sono i giorni più trafficati.

```
select month, day, count(*) as count
from flights
group by month, day
order by count desc
limit 10
```

Il giorno più frequente è quello del thanksgiving day.

Ora vogliamo sapere i giorni con massimo medio ritardo, quindi quelli in cui è meglio non prendere l'aereo.

```
select month, day, avg(dep_delay) as avg
from flights
group by month, day
order by avg desc
limit 10;
```

Poi c'è la clausola having, che seleziona solo alcuni gruppi. Per esempio vogliamo ordinare per età ma compresa tra 20 e 30. Quindi posso usare una clausola per definire i giorni affollati con più di mille voli:

```
select month, day, count(*) as count
from flights
group by month, day
having count > 1000
```

Nota: C'è differenzia sostanziale tra having e where, il primo seleziona dei gruppi il secondo seleziona delle tuple, quindi quest'ultimo non può essere utilizzato per selezionare i gruppi.

Le destinazioni popolari, cioè quelle che hanno più di 365 voli, ordinate per numero di voli in ordine decrescente:

```
select dest, count(*) as count
```

```
from flights
group by dest
having count>365
order by count desc;
```

Vogliamo usare tutte le clausole ora: selezioniamo il ritardo medio per giorno ordinato in ordine decrescente per tutti i voli nei giorni affollati del mese di luglio.

```
select month, day, count(*) as count, round(avg(dep_delay),2) as delay
from flights
where month = 7
group by day
having count>1000
order by delay desc
```

Per la creazione di questi comandi conviene iniziare per from e poi scendere perché mentalmente select lo si fa per ultimo.

Nota: la tabella weather ha avuto una modifica, invece che quattro attributi si ha riassunto in hour gli attributi che indicano il tempo di meteo.
Nota: per fare più commenti in SQL basta fare /* testo */
Dopo aver scaricato il programma SQLite abbiamo fatto una parte di verifica dei vincoli di integrità, abbiamo detto anche di verificare i vincoli di chiave esterna con anti_join. Abbiamo visto che ci sono diversi modi per sanare questa mancanza. Per farlo basta fare così:

```
# cure the violation of FKC
id_fk = anti_join(flights, planes, by = "tailnum")$id
flights = mutate(flights, tailnum = ifelse(id %in% id_fk, NA, tailnum))

# find rows violating FKC
flights %>%
  anti_join(airports, by = c("dest" = "faa")) %>%
  count(dest, sort = TRUE)

# cure the violation of FKC
id_fk = anti_join(flights, airports, by = c("dest" = "faa"))$id flights = mutate(flights, dest =
ifelse(id %in% id_fk, NA, dest))



# set time zone of time_hour of weather
library(lubridate)
tz(weather$time_hour) = "UTC" flights

%>%

  anti_join(weather, by = c("origin" = "origin", "time_hour" = "time_hour")) %>%
  count(time_hour, origin, sort = TRUE)
```

Invece il comando successivo possiamo verificare se la nostra base di dati è consistente, cioè se tutti i vincoli di integrità di chiave straniera sono verificati.

```
PRAGMA foreign_key_check
```

Se abbiamo un DB che non è consistente è sempre consigliato renderlo tale.

Ora contiamo con il nostro SQL, adesso vedremo gli operatori di insieme e poi il join, che è il più interessante:

SET OPERATOR

Sono molto semplici e alcune volte sono utili, agiscono su tabelle che hanno lo stesso schema, cioè che hanno gli stessi attributi nello stesso ordine con lo stesso tipo di dati, allora possiamo operare insiemisticamente con ***union*** (unione), ***intersect*** (intersezione) e ***except*** (differenza insiemistica)

```
-- airports that are either origins or destinations (without duplicates)
select origin
from flights
union
select dest
from flights

-- airports that are either origins or destinations (with duplicates)
select origin
from flights
union all
select dest
from flights

-- airports that are both origins and destinations
select origin
from flights
intersect
select dest
from flights

-- airports that are either origins but not destinations
select origin
from flights
except
select dest
from flights
```

Attenzione: il risultato di queste operazioni è un insieme, quindi vengono eliminati i duplicati se vogliamo mantenere i duplicati dobbiamo aggiungere la parola *all.* Questo ci porta a dire che: le tabelle in una base di dati hanno sempre le righe differenti, questo vale per tabelle originali di un DB, non per le tabelle di riposta ad una interrogazione. Quindi se poi voglio creare una tabella definitiva devo stare attento a questo.

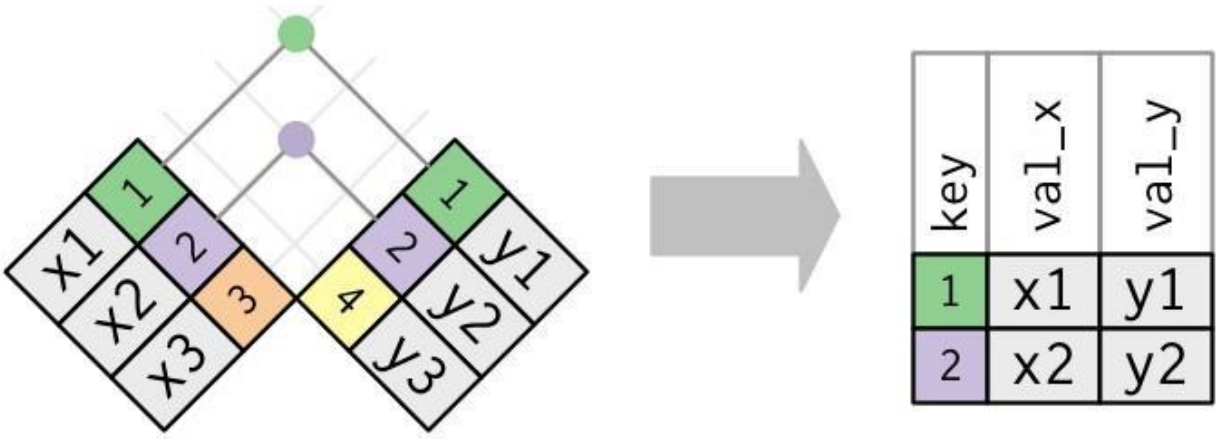
JOIN

Questo è l'operatore più importante. Nel modello relazionale si usano solo le tabelle, a differenza del modello entità relazioni, quindi sostanzialmente suddivido la mia informazioni in tante tabelle quante sono le entità, legando queste tabelle tramite le parole esterne o straniere. Una volta che ho separato/normalizzato la mia informazioni in tabelle, devo fare un join (collegare/unire) le tabelle per estrapolare informazioni dalle due tabelle. (es. tabelle studente corso e esame, se voglio sapere quali sono i nomi degli studenti che hanno superato un certo corso con un qualche voto, chiaramente dobbiamo "unire" queste tabelle per giungere all'informazione richiesta).

Supponiamo di voler estrarre tutti i voli che hanno volato con un aeroplano costruito dalla BOEING, Se si vanno a vedere i due dataset planes e flights è abbastanza chiaro che dobbiamo unire queste due tabelle, nel senso che in flights troviamo le informazioni sui voli, in particolare il tailnum, in Planes troviamo il codice dell'aeroplano e il costruttore, quindi è chiaro che non possiamo usare una sola di queste due tabelle. Per connettere queste due tabelle si fa un prodotto cartesiano, selezionando solo le tuple che ci servono. Immaginiamo di fare un prodotto cartesiano di due tabelle, il risultato sarà una tabella costituita da righe costituite da una riga di flights e una di planes. Se flights è costruita da n righe e Planes da m allora il prodotto cartesiano sarà costituito da righe. Formalmente questo che il prodotto cartesiano:

$$A \, X \, B = \{(x \, , y) \mid x \in A \, , y \in B\}$$

Le uniche righe di questo prodotto cartesiano che ci interessano sono quelle per cui la chiave esterna di flights combacia con la chiave primaria di planes. Join fa proprio questo: fa il prodotto cartesiano e connette le righe che hanno questo legame. Quindi quello che dovremo scrivere è questo:




```
-- flights that flew with a plane manufactured by BOEING
select flights.id, flights.tailnum, planes.manufacturer
from flights, planes
where flights.tailnum = planes.tailnum and planes.manufacturer = "BOEING"
```

I comandi dentro li abbiamo già visti, l'unica differenza è che usiamo due tabelle. Notare che essendo *tailnum* ripetuto nelle due tabelle per specificare di quale stiamo trattando si scrive *Nometabella.NomeAttributo*, (es. *flights.tailnum*). Genericamente l'operazione di join tra due tabelle R e S è una congiunzione (secondo l'operatore Booleano) di una condizione atomica del tipo $A \theta B$, dove A è un attributo di R e B è un attributo di S e θ è un operatore di comparazione. Solitamente, ma non sempre, A è una chiave primaria e B è una chiave esterna riferente ad A e θ è l'operatore di uguaglianza =.

Un join non è nient'altro che due *for loop* annidati: uno che scandisce la prima tabella e uno che scandisce una seconda tabella.

```
-- flights that flew to a destination with an altitude greater than 6000 feet
sorted by altitude
select flights.id, flights.dest, airports.name, airports.alt
from flights, airports
where flight.dest = airports.faa and airports.alt > 6000
order by airpots.alt
```

Attenzione: nel *where* bisogna anche dichiarare il collegamento tra le due tabelle, cioè tra chiave esterna e chiave primaria delle due tabelle.

Nulla ci vieta di fare join di più tabelle, di solito si fa di due tabelle, più raramente in tre tabelle e ancora più raramente più di tre tabelle, poiché essendo due cicli for è una operazione molto costosa. Proprio perché il prodotto cartesiano ha come cardinali il prodotto delle cardinalità:

$$| A \, X \, B | = | A | \, | B |$$

In questo caso sottostante vediamo questa casistica di tre tabelle:

```
-- flights that took off with a plane with 4 engines and a visibility lower than
3 miles
select flights.id, planes.engines, weather.visib
from flights, weather, planes
where flights.time_hour = weather.time_hour and flights.origin = weather.origin
      and flights.tailnum = planes.tailnum and weather.visib < 3  and
planes.engines = 4
limit 10
```

Notiamo che l'operazione *where* è molto lunga perché bisogna legare le chiavi di tutte e tre le tabelle. A questo punto ogni riga avrà informazioni riguardanti il volo, il corrispondente aereo e il corrispettivo meteo.

Ancora una volta la cosa giusta da fare è partire dallo schema relazionale scritto a priori, in modo tale da non sbagliare il collegamento tra le diverse chiavi. Quindi due cose: prima cosa bisogna capire quali sono le tabelle per un join e seconda cosa capire come collegarle. Chiaramente qui capiamo l'importanza delle relazioni concettuali e come le abbiamo costruite nel modello relazionale. Nulla vieta di fare il join sulla medesima tabella, per esempio supponiamo di voler calcolare tutti i cammini di volo di lunghezza due. Per cammino di volo intendiamo una cosa di tipo $X -> Y -> Z$, dove X e Y e Z sono tre luoghi, sostanzialmente degli scali.

```
-- flight paths of length 2 (X --> Y --> Z)
select distinct f1.origin, f1.dest, f2.dest
from flights as f1, flights as f2
where f1.dest = f2.origin
```

Sostanzialmente abbiamo voli che vanno da a a b, $a -> b$ e voli che vanno da c a d, $c -> d$, volgiamo trovare quelli che hanno $b = c$. Vediamo ora con lunghezza 3:

```
-- flight paths of length 3 (X --> Y --> W --> Z)
select distinct f1.origin, f1.dest, f2.dest, f3.dest
from flights as f1, flights as f2, fights as f3
where f1.dest=f2.origin and f2.dest=f3.origin
```

Quest'ultimo comando non da alcun risultato perché non ce ne sono, nessuna paura. Potrei trovare tutti i cammini arbitrari? Cioè di lunghezza arbitraria? Per esempio si da una città di partenza e trovare tutte le destinazioni che posso raggiungere da Roma con un arbitrario numero di scali. Ci servirebbe un'interrogazione infinita, in realtà vedremo una spiegazioni formale. Non si può fare. Ci sono dei bisogni informativi che non si possono scrivere. Questo è un limite del linguaggio, ed è perfettamente normale che un linguaggio abbia dei limiti. Vediamo un'altra query che fa due join e risulta essere abbastanza complicata: vogliamo tutti i voli che hanno volato in due posti con grande differenza di altitudine, cioè che la differenza di altitudine e di destinazione sia più di 6000 piedi:

```
-- flights with destination and origin airports with an altitude difference of
more than 6000 feet
select flights.id, airports2.alt, airports1.alt, (airports2.alt - airports1.alt)
as altdiff
from flights, airports as airports1, airports as airports2
where flights.origin = airports1.faa and flights.dest = airports2.faa and
altdiff > 6000
```

L'operazione di join è talmente importante che ha una sintassi specifica.

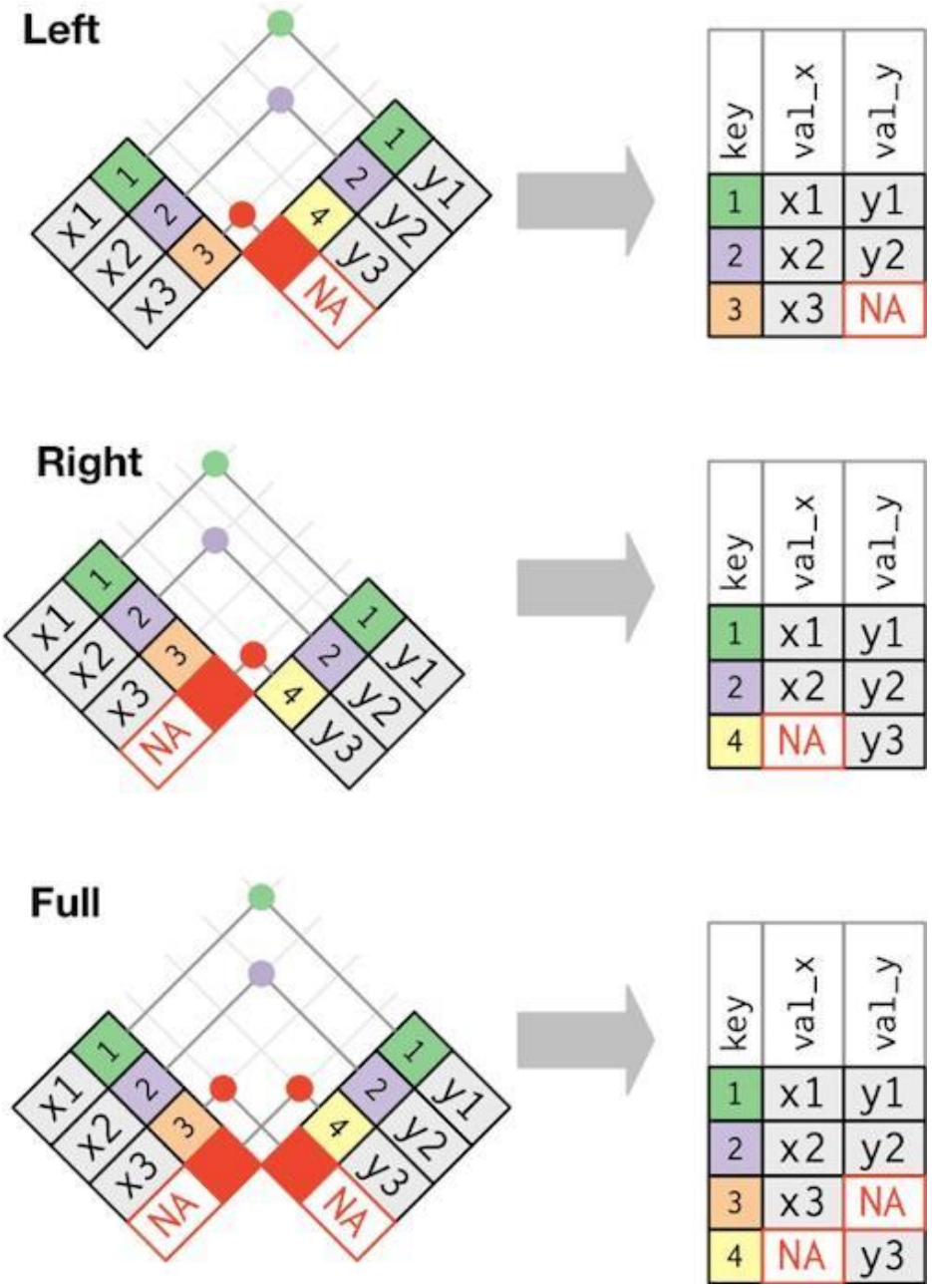
```
-- flights that flew with a plane manufactured by BOEING
select flights.id, flights.tailnum, planes.manufacturer
from flights join planes on flights.tailnum = planes.tailnum
where planes.manufacturer = "BOEING"
```

Questo modo è consigliato per scrivere il join, e ha alcuni vantaggi:

- È più chiaro, poiché separa la condizione di join da altre condizioni
- Permette di distinguere tra diversi tipi di join: ***inner join***, ,***left outer join***, ***right outer join*** e ***full outer join***.
    1. **Inner join:** filtra le tuple che soddisfano le condizioni di join (clausola usata fino ad ora)
    2. **Left outer join:** oltre a filtrare le tuple che soddisfano la condizione di join raccoglie anche le tuple della tabella di sinistra che non trovano una corrispettiva tupla della tabella di destra (queste tuple avranno valore NULL nella tabella di destra)
    3. **Right outer join:** inversione del *left join outer*, cioè prende le tuple che soddisfano la condizione di join e le tuple della tabelle di destra che non hanno un corrispettivo nella tabella di sinistra (restituisce sempre il NULL)

4. **Full outer join:** unione delle due precedenti, cioè prende tutte le tuple che soddisfano la condizione di join e tutte le tuple della composte da righe della tabella di destra e di sinistra che non trovano un corrispettivo (restituendo sempre NULL).

Immaginiamo che una tupla non abbia la sua corrispondente che soddisfa la selezione nella creazione del piano cartesiano (es. un volo con tailnum che non esiste nella tabella planes). Queste tuple non vengono selezionate nel join, quindi non le metto nell'output. Questo tipo di join si chiama **inter join**, cioè che seleziona solo le righe che soddisfano la condizione booleana. Il *left join* invece fa fuoriuscire anche questi voli. Il *right join* invece fa il contrario, cioè va a prendere anche gli aeroplani che non hanno associato un volo. Il *full join* fa l'unione di *left* e *right*. Vediamo uno schema che ci illustra la differenza tra i quattro tipi di join:



Vediamo un esempio:

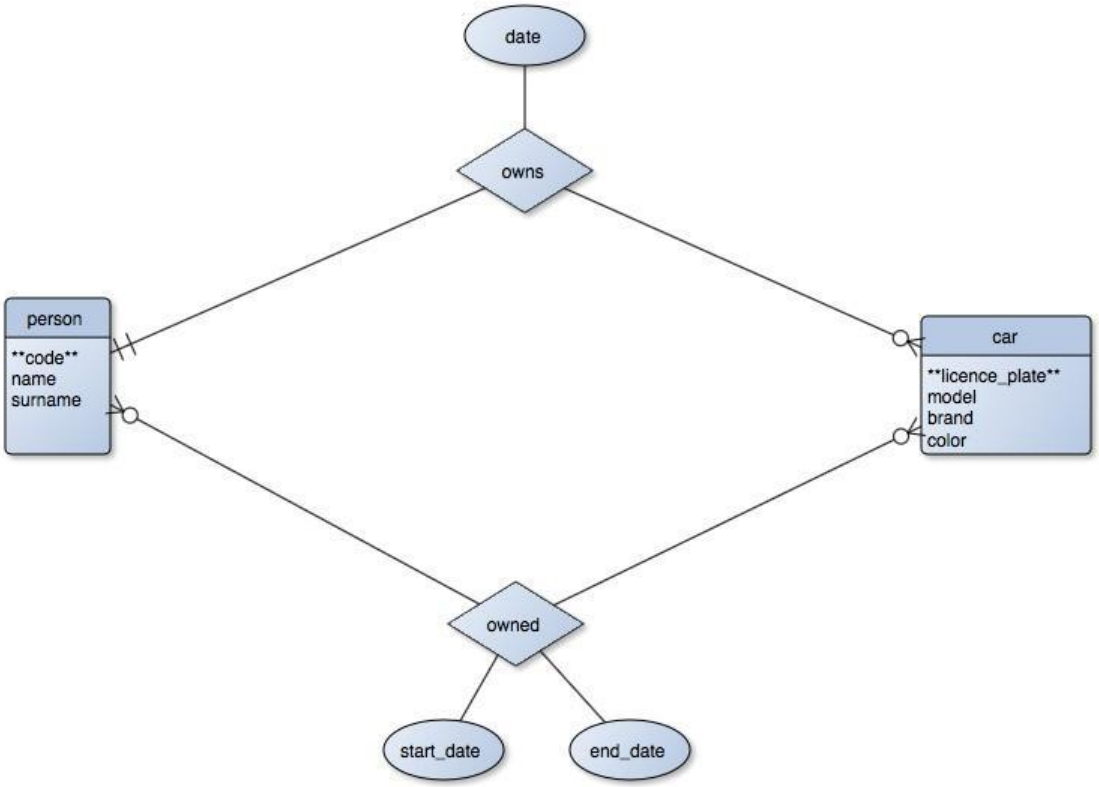
```
-- flights and the corresponding plane manufactured, including flights that have
no match in planes (there are some)
```

```
select flights.id, flights.tailnum, planes.manufacturer
from flights left join planes on flights.tailnum = planes.tailnum

-- inner count (only those with a match)
select count(*)
from flights join planes on flights.tailnum = planes.tailnum

-- outer count (all flights)
select count(*)
from flights left join planes on flights.tailnum = planes.tailnum
```

A seconda dell'aggiornamento del DB il risultato sarà diverso, se si sta usando il DB che rispetta i vincoli di integrità allora i risultati saranno uguali, mentre se si usa il DB senza correzioni allora i risultati saranno diversi. Il *left join* esiste, mentre il *right* e il *full* non esistono, quindi bisogna simularli:

```
-- simulate a right join on planes and flights using left join
select  planes.manufacturer, flights.id, flights.tailnum
from planes left join flights on flights.tailnum = planes.tailnum

-- simulate a full join on planes and flights using left join and union
select flights.id, flights.tailnum, planes.manufacturer
from planes left join flights on flights.tailnum = planes.tailnum
union all
select flights.id, flights.tailnum, planes.manufacturer
from flights left join planes on flights.tailnum = planes.tailnum
Where flights.tailnum is null;
```

Con questo finiamo SQL. Concludiamo con una esercitazione conclusiva di SQL.
We are going to create in SQLite a database corresponding to the above logical schema:



> person(<u>tax_code</u>, name, surname)
> car(<u>license_plate</u>, model, brand, color, owner, date)
> owned(<u>person</u>, <u>car</u>, start_date, end_date)
> car(owner) –> person(tax_code)
> owned(person) –> person(tax_code)
> owned(car) –> car(license_plate)

1. create a database called [redacted] in SQLite
2. create tables with primary and foreign key constraints using SQL create table
3. insert data into tables using SQL insert
4. enable foreign integrity constraints with PRAGMA foreign_keys and check them with PRAGMA foreign_key_check
5. create indices on primary and foreign keys
6. backup the database
7. violate primary and foreign key constraints with SQL insert, delete, and update
8. restore the database from the backup
9. write and run some queries. Try to use all the SQL clauses (select, from, where, group by, having, order by, join)

```
cd /Users/andreapesce/Desktop/SQLcar
SQLite3 cars
. separator ,
. mode csv

CREATE TABLE person (
 tax_code VARCHAR(16),
 name VARCHAR(60),
 surname VARCHAR(60),
 primary key (tax_code)
);

CREATE TABLE cars (
 license_plate VARCHAR(6),
 model VARCHAR(60),
 brand  VARCHAR(60),
 color  VARCHAR(60),
 owner  VARCHAR(16),
 date TEXT(10),
 primary key (license_plate)
 foreign key (owner) references person(tax_code)
);

CREATE TABLE owned (
 person VARCHAR(16),
 car VARCHAR(6),
 start_date TEXT(10),
 end_date TEXT(10),
 primary key (person, car),
 foreign key (person) references person(tax_code)
 foreign key (car) references cars(license_plate)
);

insert into person values ('NFKAPO86J95T593L', 'GIANLUCA', 'IACUBINO');
insert into person values ('NFKDPO86J95T593L', 'MICHELE', 'IACUBINO');
insert into person values ('NDLAPO86J95T593L', 'PIPPO', 'BAUDO');
insert into person values ('LAKAPO86J95T593L', 'ANDREA', 'PESCE');
insert into person values ('NFKKPO87J95T593L', 'LUCA', 'BURATTO');

insert into cars values ('AS7899','POLO',
'WOLSWAGEN','BLU','NFKAPO86J95T593L','10-03-2007');
insert into cars values ('AS7880','GOLF',
'WOLSWAGEN','GRIIGO','NFKDPO86J95T593L','10-03-2007');

insert into owned values ('NFKDPO86J95T593L' , 'AS7899' ,
'13-03-2000','10-03-2007');
```

```
PRAGMA foreign_key_check;
```

```
PRAGMA foreign_keys = ON;



create unique index person_index on person(tax_code);
create unique index cars_index on cars(license_plate);
create unique index owned_person on owned(person);
create unique index owned_car on owned(car);

.backup cars_backup_20180319
```

Tutto quello che abbiamo fatto lo abbiano fatto tramite un DBMS adesso vedremo come si fa in R. Sfrutteremo tre pacchetti ***DBI***, ***RSQLite*** e ***dbplyr***. Quasi tutto quello che abbiamo visto potremo rifarlo su R. Visto che sappiamo usare SQL si consiglia di usare SQL per fare le query, per lavorare su DataSet si può usare R. Questo esercizio va fatto dopo la creazione del DB e delle tabelle con SQLite.

```
## Load libraries
library(DBI) library(dplyr)
library(dbplyr)
library(nycflights13)

# Connect to the database
nyc <- dbConnect(RSQLite::SQLite(), "nycflights13")

# If you just need a temporary database, use either "" (for an on-disk database) or ":memory:" (for a in-memory database). This database will be automatically deleted when you disconnect from it.



# Add surrogate key to flights flights <-
  flights %>%
  arrange(year, month, day, sched_dep_time, carrier, flight) %>%
  mutate(id = row_number()) %>%
  select(id, everything())


# cure the violation of FKC
id_set_planes = anti_join(flights, planes, by = "tailnum")$id id_set_airports = anti_join(flights, airports, by =
c("dest" = "faa"))$id flights = mutate(flights, tailnum = ifelse(id %in% id_set_planes, NA, tailnum),
                          dest = ifelse(id %in% id_set_airports, NA, dest))


Possiamo scrivere i dati direttamente con la funzione sottostante scriverli da un datframe, non occorre più fare un csv. In un colpo solo popoliamo la tabella flights con i dati del dataset flights gestendo i casi NA o NULL in modo corretto.
# write data frames into database tables (if necessary)
dbWriteTable(nyc, "flights", flights) dbWriteTable(nyc,
"airports", airports) dbWriteTable(nyc, "planes", planes)
dbWriteTable(nyc, "weather", weather)
dbWriteTable(nyc, "airlines", airlines)

# list tables
dbListTables(nyc)

# list fields of a table dbListFields(nyc,
"flights")
Prima creiamo la query e poi la applichiamo con il comando successivo, e il risultato è una tabella. Qui una query è una stringa di R, quindi si scrive tra virgolette.
```

```
# query with SQL
query1 =
"select id, month, day, sched_dep_time, carrier, flight number from flights
where month = 12 and day = 25 limit
10"

dbGetQuery(nyc, query1)

query2 =
"select flights.id, airports2.alt, airports1.alt, (airports2.alt - airports1.alt) as altdiff
from flights, airports as airports1, airports as airports2
where flights.origin = airports1.faa and flights.dest = airports2.faa and altdiff > 6000 limit 10"

dbGetQuery(nyc, query2)
```

Possiamo scriverle query in dplyr

```
# Alternatively, query with dplyr using dbplyr package

# get a reference to the table flights flights_db <-
tbl(nyc, "flights")

# run a query in dplyr flights_db
%>%
    group_by(dest) %>%
    summarise(delay = mean(dep_time))
```

Abbiamo così la conversone in SQL della nostra interrogazione. Cioè nellOutput di R viene mostrata una stringa che si può utilizzare in SQL.

```
# show the SQL statement flights_group_db <-
flights_db %>%
    group_by(dest) %>%
    summarise(delay = mean(dep_time))

flights_group_db %>% show_query()
```

Possiamo anche eseguire la query in altro modo che può esser utile se sappiamo a priori che il risultato della query è molto grosso lavorando a blocchi con un ciclo while. Questa è una tecnica che si usa molto spesso.

```
# If you run a query and the results don't fit in memory, you can use dbSendQuery(), dbFetch() and dbClearResults()
to retrieve the results in batches.
rs <- dbSendQuery(nyc, 'select * from flights limit 100')
while (!dbHasCompleted(rs)) { df <-
    dbFetch(rs, n = 10) print(nrow(df))
}
dbClearResult(rs)


# disconnect the database
dbDisconnect(nyc)
```

Per non avere il DB fisico sul disco

```
# remove the database (if necessary)
unlink("nycflights13")
```

Questa è una tecnica per accedere a dati di grosse dimensioni che contiene diverse entità e relazioni, si conviene progettare una base di dati. Per far ciò bisogna avere cura e concentrazione. Nel lungo termine questa cosa ha ovviamente i suoi frutti.

2.4. Perché usiamo SQL?

La risposta è data da una teoria delle base dei dati relazionali di cui vedremo una piccola fetta. Come dicevamo la forza del modello relazionale è dovuta al fatto che la nozione di tabella è sia molto pratica, ma anche teorica, cioè definita tramite l'algebra con delle operazioni su tabelle, quindi una algebra relazionale. Una **algebra** è un insieme di operazioni su un dominio, per esempio una algebra elementare con la somma +, sottrazione -, prodotto e divisione / in $\mathbb{R}$ per esempio. Questa algebra ha come input numeri reali e come output un numero reale. Allo stesso modo possiamo costruire una algebra delle relazioni. In cui il nostro dominio sono le tabelle o relazioni e le operazioni possibili sono quelle che SQL ha introdotto, fondamentalmente per l'algebra relazionale sono:

- Proiezione: si selezionano alcune colonne partendo da una tabella e corrisponde alla clausola *select*.
- Selezione: si selezionano alcune righe che soddisfano certe condizioni e corrisponde alla clausola *where*
- Join: va a combinar due tabelle facendo il prodotto cartesiano e filtrandole.
- Unione: è l'operazione che prende l'unione di due tabelle.
- Differenza: è l'operazione che prende la differenza tra due tabelle.
- Rinomina: serve per rinominare gli attributi della tabella, e corrisponde alla clausola as nella clausola select

Queste sono le operazioni base dell'algebra relazionale. L'algebra relazionale è un linguaggio operazionale, a differenza di SQL che è dichiarativo, o procedurale. Cioè un'espressione ci dice come arrivare ad un risultato e non ci da il risultato. In un certo senso possiamo rifrasare la domanda del titolo in "perché algebra lineare?", siamo così passati intuitivamente ad una domanda più teorica. Quando abbiamo un linguaggio le due cose che ci possiamo chiedere è: qual è la sua **espressività** e qual è la sua **complessità**.

Come abbiamo visto precedentemente in alcuni linguaggi non si può scrivere tutto quello che vogliamo. Questo limite entra nel concetto di espressività. I linguaggi possono esprimere un certo numero di concetti.

La complessità invece è interpretabile come costo di elaborazione del programma, il cui costo viene definito con due parametri: il **tempo** e lo **spazio**. Quindi siamo interessati a quanto tempo impiegano i programmi a terminare e quanta memoria utilizzano. Il tempo risulta più importante dello spazio. Il tempo è più critico perché lo spazio lo si può riutilizzare e il tempo no. Più un linguaggio è espressivo più sarà complesso. La complessità si calcola nel caso peggiore *worst case complexity* tramite una funzione.

Informalmente l'algebra relazionale ha il potere espressivo del calcolo dei **predicati di primo ordine**, che è una logica molto famosa.

Per la complessità computazionale, il problema della valutazione di una espressione dell'algebra relazionale con riferimento ad un DB è completo nella classe di complessità PSPACE, cioè nella classe di problemi risolvibili usando uno spazio polinomiale. Ad esempio: $f(u) = 3u^3 + 5$. Questo sembra essere uno brutta notizia, poiché non esistono algoritmi polinomiali per questo problema. Infatti, la valutazione della complessità di una espressione è di forma esponenziale nella dimensione della espressione o query e polinomiale nella dimensione del DB. Solitamente la complessità è espressa tramite due parametri n e k che rappresentano la pesantezza del DB e la complessità della query. Essendo n>>k, allora spesso la funzione di complessità è data da $\phi(n, k) = n^k$, ed essendo k molto più piccolo di n può esser approssimato ad un polinomio e quindi si torna al caso precedente. Nel senso che la grandezza del DB è molto più grande della dimensione della query, tanto da considerare la grandezza della query costante e non più come un parametro di complessità. Di conseguenza fissando la dimensione della query come costante e riferendosi solo alla complessità dei dati, possiamo avere: il problema della valutazione dell'algebra relazionale è nel PTIME, la classe di problemi risolvibili in prodotti polinomiali (polynomial time). In particolare, il problema è situato nella classe LOGSPACE, una sottoclasse di PTIME, a cui corrispondono i problemi che possono essere risolti nello stazione logaritmico.

La classe LOGSPACE è una low class nella gerarchia delle classi della complessità computazionale. In particolare, il problema contenuto in questa classe può essere risolto in vie parallele.

L'ultima bella proprietà dell'algebra relazionale è l'ottimizzazione: ogni espressione di SQL può essere trasformata, con complessità polinomiale tenendo conto della sua grandezza, in una espressione equivalente dell'algebra relazionale e viceversa.

Questa proprietà caratterizza l'algebra relazione come la controparte procedurale di SQL. Si noti che la trasformazione da SQL all'algebra è efficiente, nel senso informatico del termine. Questo risultato apre le porte ad un serie di ottimizzazioni che può essere utilizzata durante la valutazione di una query di SQL. Una query di SQL, per essere valutata, è prima trasformata in un espressione dell'algebra relazionale. Questa

espressione è poi riscritta in una qualche equivalente, ma ottimizzata, versione, tenendo conto di qualche misura di complessità. L'operazione più costosa dell'algebra relazionale è il join e la complessità di tale operatore dipende dalla cardinalità delle tavole nell'argomento. Infatti a causa del join la complessità diventa molto alta perché essendo dei cicli for annidati il k diventerebbe molto grande. Se facciamo un join su h tabelle allora i cicli saranno h-1, a questo punto il secondo parametro sarà dell'ordine di 2^{h-1} . lo scopo di riscrivere l'espressione relazionale è quello di minimizzare il numero di operatori join e di eseguire questi operatori in piccole tabelle. Quest'ultimo risultato è ottenuto "spingendo" le selezione antecedente dentro il join con lo scopo di filtrare le tabelle prima di sottoporle alla operazione di join.

Si può prendere la query dichiarativa in SQL e si può trasformare con costo polinomiale all'algebra lineare e riscrivendo l'espressione in diversi modi in modo da renderla efficiente e poi eseguirla. Questo è quello che fa un modulo di DBMS: l'**ottimizzatore**. Nel senso che l'ottimizzatore utilizza semplificazioni presenti nell'algebra lineare che danno lo stesso risultato diminuendo la "pesantezza" delle tabelle.

In conclusione, le motivazioni dietro al successo di SQL sono la sua espressività, una buona complessità e l'abilità di risolvere efficientemente le query, cioè si presta ad essere ottimizzato.

2.5 Algebra Relazionale e Calcoli

Introdurremo in modo un po' più formale il modello relazionale e l'**algebra relazionale** e vedremo il **calcolo relazionale dei domini**, con lo scopo di capire il perché la nostra interrogazione vista precedentemente non possa esser applicata in SQL. Per rammentare al lettore l'interrogazione riguardava una query in cui il numero di scali di viaggio era arbitrario.

Le base di dati non sono solo uno strumento, un programma o un software, ma sono una **teoria**. Vedremo un accenno di questa teoria.

2.5.1 Modello Relazionale

Informalmente un modello relazionale definisce tutto con solo la relazione, cioè la tabella. Definiamo uno **schema di relazione** R come l'insieme dei suoi attributi:

$$s\,ch\,e\,m\,a\,(R) = \{A_1, \ldots, A_m\}$$

che descrive le proprietà della relazione e dove $A_1, \ldots, A_m$ sono gli attributi. E lo **schema di un database** è l'insieme degli schemi delle relazioni.

Ogni **attributo** A è associato a un **dominio di valori**, denotato con $D\,(A)$. Un dominio è atomico se risulta essere un insieme non divisibile di valori che non ha una struttura interna. (Uno schema di relazioni è nel First Normal Form (1NF) se tutti i domini degli attributi sono atomici. Uno schema di database è nel 1NF se ogni schema di relazioni è nel 1NF.)

Una **tupla** su una schema di relazioni R, con $s\,ch\,e\,m\,a\,(R) = \{A_1, \ldots, A_m\}$ è un elemento del prodotto cartesiano:

$$D\,(A_1) \times D\,(A_2) \times \ldots \times D\,(A_m)$$

Una **relazione** è un insieme finito di tuple e un **database** è un insieme finito di relazioni. È importante ricordare che una relazione e una DB sono insiemi finiti, solo un numero finito di informazioni possono essere salvati su un computer.

2.5.2 Algebra Relazionale

L'**algebra relazionale** è una collezione di operatori; ogni **operatore** prende come input o una singola relazione o più relazioni e come output ha una singola relazione, cioè come suo risultato. Una **query relazionale** è una combinazione di un numero finito di operatori relazionali. In questo senso una query è **procedurale**, poiché specifica l'ordine in cui gli operatori compaiono nella query e vengono valutati. La

corrispondente parte dichiarativa dell'algebra relazionale, come il calcolo relazionale dei domini, sarà definita dopo.

L'algebra relazionale è sempre stata inventa da Codd nel medesimo articolo, anche il calcolo relazionale.

Lo stile di presentazione è il seguente: per ogni operatore dell'algebra relazionale daremo prima una definizione informale a parole, poi una definizione formale e poi la definiremo tramite il linguaggio dplyr. Lavoreremo sul solito database.

```
library(nycflights13)
library(dplyr)
```

UNIONE

L'unione, la differenza e l'intersezione sono **operatori binari** che prendono due relazioni con lo stesso schema: prendono due relazioni su un comune schema di relazione e restituiscono una relazione sul medesimo schema. L'**unione** di due relazioni r_1 e r_2 su uno schema di relazione è un insieme di tuple che sono sia in r_1 che in r_2:

$$r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$$

La funzione *union*() implementa l'operatore di unione in dplyr:

```
r1 = filter(airlines, row_number() <= 5)
r2 = filter(airlines, row_number() <= 10, row_number() >= 4)
r1
```

```
## # A tibble: 5 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
## 5 DL      Delta Air Lines Inc.
```

```
r2
```

```
## # A tibble: 7 x 2
##   carrier name
##   <chr>   <chr>
## 1 B6      JetBlue Airways
## 2 DL      Delta Air Lines Inc.
## 3 EV      ExpressJet Airlines Inc.
## 4 F9      Frontier Airlines Inc.
## 5 FL      AirTran Airways Corporation
## 6 HA      Hawaiian Airlines Inc.
## 7 MQ      Envoy Air
```

```
union(r1, r2)
```

```
## # A tibble: 10 x 2
##    carrier name
##    <chr>   <chr>
##  1 MQ      Envoy Air
##  2 HA      Hawaiian Airlines Inc.
##  3 FL      AirTran Airways Corporation
##  4 F9      Frontier Airlines Inc.
##  5 EV      ExpressJet Airlines Inc.
##  6 DL      Delta Air Lines Inc.
##  7 B6      JetBlue Airways
##  8 AS      Alaska Airlines Inc.
##  9 AA      American Airlines Inc.
## 10 9E      Endeavor Air Inc.
```

DIFFERENZA

La **differenza** tra due relazioni r_1 e r_2 su uno schema di relazione R è l'insieme delle tuple che stanno in r_1 ma non in $r2$;

$$r_1 \setminus r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$$

La funzione corrispondente è *setdiff*():

```
r1
```

```
## # A tibble: 5 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
## 5 DL      Delta Air Lines Inc.
```

```
r2
```

```
## # A tibble: 7 x 2
##   carrier name
##   <chr>   <chr>
## 1 B6      JetBlue Airways
## 2 DL      Delta Air Lines Inc.
## 3 EV      ExpressJet Airlines Inc.
## 4 F9      Frontier Airlines Inc.
## 5 FL      AirTran Airways Corporation
## 6 HA      Hawaiian Airlines Inc.
## 7 MQ      Envoy Air
```

```
setdiff(r1, r2)
```

```
## # A tibble: 3 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
```

INTERSEZIONE

L'**intersezione** tra due relazioni r_1 e r_2 su uno schema di relazione R è l'insieme delle tuple che stanno sia in r_1 che in r_2

$$r_1 \cap r_2 = \{t \mid t \in r_1 \wedge t \in r_2\}$$

E la funzione in dplyr è *intersect*().

```
r1
```

```
## # A tibble: 5 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
```

```
## 5 DL      Delta Air Lines Inc.
```

```
r2
```

```
## # A tibble: 7 x 2
##  carrier name
##  <chr>   <chr>
## 1 B6      JetBlue Airways
## 2 DL      Delta Air Lines Inc.
## 3 EV      ExpressJet Airlines Inc.
## 4 F9      Frontier Airlines Inc.
## 5 FL      AirTran Airways Corporation
## 6 HA      Hawaiian Airlines Inc.
## 7 MQ      Envoy Air
```

```
intersect(r1, r2)
```

```
## # A tibble: 2 x 2
##  carrier name
##  <chr>   <chr>
## 1 B6      JetBlue Airways
## 2 DL      Delta Air Lines Inc.
```

Possiamo definire l'intersezione usando la differenza insiemistica?

$$r_1 \cap r_2 = r_2 \setminus (r_2 \setminus r_1)$$

In questo modo vediamo come l'intersezione in realtà risulta essere ridondante. Per questo solitamente si parla solo di unione e differenza

PROIEZIONE

La **proiezione** di una relazione *r* su uno schema di relazione *R* e su un insieme di attributi X incluso nello *schema*(*R*) è l'insieme delle tuple risultanti da una proiezione di ogni tupla *r* negli attributi di *X*:

$$\pi(r, X) = \{t[X] \mid t \in r\}$$

dove $t[X]$ è la proiezione della tupla *t* negli attributi in *X*, cioè la tuple che contiene solo i valori di *t* per gli attributi in *X*. Sostanzialmente è il s*elect* di SQL. Ad esempio:

tupla

A	B	C

proiezione $\pi(t, \{A, B\}) =$

A	B

La funzione è *select*().

```
select(planes, tailnum, manufacturer)
```

```
## # A tibble: 3,322 x 2
##    tailnum manufacturer
##    <chr>   <chr>
##  1 N10156  EMBRAER
##  2 N102UW  AIRBUS INDUSTRIE
##  3 N103US  AIRBUS INDUSTRIE
##  4 N104UW  AIRBUS INDUSTRIE
##  5 N10575  EMBRAER
##  6 N105UW  AIRBUS INDUSTRIE
```

```
##  7 N107US  AIRBUS INDUSTRIE
##  8 N108UW  AIRBUS INDUSTRIE
##  9 N109UW  AIRBUS INDUSTRIE
## 10 N110UW  AIRBUS INDUSTRIE
## # ... with 3,312 more rows
```

Notiamo che la proiezione cattura la semantica della quantificazione esistenziale. Per esempio, nella query precedente abbiamo recuperato il valore delle tuple per il tailnum e il manufacturer così come esistono valori per le variabili year, type, ecc. per queste tuple.

L'algebra relazionale è una algebra su relazioni cioè insiemi, gli insiemi non contengono duplicati. Facendo la proiezione essa può portare a dei duplicati, cioè ci possono essere tuple che hanno tutti gli attributi diversi ma un sottoinsieme di valori uguali. dplyr non elimina i duplicati nel risultato. (In arancione evidenziata la proiezione).

a	a	b
a	a	c

Cioè in R:

```
d = data.frame(x = c("a", "a", "b"), y = c(1, 2, 3))
select(d, x)
```

```
##   x
## 1 a
## 2 a
## 3 b
```

Nell'algebra relazionale invece non ci sono duplicati, perché nell'algebra non ci sono mai duplicati.
$\pi(d, x) = \{a, b\}$.

SELEZIONE

È un po' più complessa perché è una **operazione unaria** per la quale bisogna introdurre i predicati booleani. La **selezione** di una tupla da una relazione r rispettando una formula di selezione F è il sottoinsieme di tuple di r che soddisfano la **formula** F. Una formula di selezione è definita ricorrentemente come segue:

1. Una **semplice formula** di selezione su uno schema di relazione R è o una espressione della forma $A = a$ o una espressione della forma $A = B$, dove $A, B \in schema(R)$ e $a \in D(A)$.
2. Una formula di selezione su R è una espressione **ben definita** composta da una o più semplici formule di selezione R insieme a una $\wedge(and)$, $\vee(or)$, $\neg(not)$ e connessione logica Booleana:
   parentesi.

Per esempio:

$$A = B \wedge (C = c_1 \vee C = c_2)$$

Informalmente una tupla, t, implica logicamente una formula, F, se la tupla soddisfa F. Formalmente, una tupla t *che* implica logicamente F, scritto $t \vDash F$, è definita ricorrentemente, come segue:

1. $t \vDash A = a \; set[A] = a \; vero$
2. $t \vDash A = B \; set[A] = t[B] \; vero$
3. $t \vDash F_1 \wedge F_2 \; set \vDash F_1 \; vero \; e \; t \vDash F_2 \; vero$
4. $t \vDash F_1 \vee F_2 \; set \vDash F_1 \; vero \; o \; t \vDash F_2 \; vero$
5. $t \vDash \neg F_1 \; set \vDash F_1 \; falso$

La selezione, , applicata ad una relazione su uno rispettando una formula di
schema di relazione *R*
selezione *F* su *R* è definita da:

$$\sigma(r, F) = \{t \mid t \in r \wedge t \models F\}$$

La funzione *filter*() implementa l'operatore di selezione in dplyr:

```
filter(planes, manufacturer == "EMBRAER" & engines == 2)
```

```
## # A tibble: 299 x 9
##    tailnum  year type       manufacturer model engines seats speed engine
##    <chr>   <int> <chr>      <chr>        <chr>   <int> <int> <int> <chr>
##  1 N10156   2004 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  2 N10575   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  3 N11106   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  4 N11107   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  5 N11109   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  6 N11113   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  7 N11119   2002 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  8 N11121   2003 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
##  9 N11127   2003 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
## 10 N11137   2003 Fixed win… EMBRAER      EMB-1…      2    55    NA Turbo…
## # ... with 289 more rows
```

C'è una naturale corrispondenza $\wedge$, $\vee$, $\neg$ presenti in una formula di
tra i connettivi logici Booleani

selezione e l'insieme degli operatori $\cup$, $\cap$, $\setminus$ rispettivamente:

1. $\sigma(r, F_1 \vee F_2) = \sigma(r, F_1) \cup \sigma(r, F_2)$
2. $\sigma(r, F_1 \wedge F_2) = \sigma(r, F_1) \cap \sigma(r, F_2)$
3. $\sigma(r, \neg F) = r \setminus \sigma(r, F)$

JOIN

Informalmente, il **join naturale** di due relazioni r_1 su uno schema di relazione R_1 e r_2 su uno schema di relazioni R_2, con $schema(R_1) \cap schema(R_2)$ l'insieme degli attributi X, è la relazione contenente tuple che provengono dalla concatenazione di tutte le tuple r_1 con tutte le tuple r_2 tali che entrambe delle quali hanno gli stessi X-valori. Gli attributi in X sono chiamati gli attributi di join di R_1 e R_2.

Formalmente, il join naturale, $r_1 \bowtie r_2$, di due relazioni r_1 su uno schema di relazione R_1 e r_2 su uno schema di relazione R_2 è una relazione su uno con schema di relazione R

$schema(R) = schema(R_1) \cup schema(R_2)$ definito da:

$$r_1 \bowtie r_2 = \{t \mid t[schema(R_1)] \in r_1 \wedge t[schema(R_2)] \in r_2\}$$

X={B}

Relazione T1

Relazione T2

B	D

Condizione di join: $T1.B = T2.B$
La condizione di join non vien scritta, ma è implicita.

$T1 \bowtie T2$ $\xrightarrow{Porta\ a}$

1. Cos'è $r_1 \bowtie r_2$ se $sch em a(R_1) = sch em a(R_2)$? L'intersezione
2. Cos'è $r_1 \bowtie r_2$ se $sch em a(R_1) \cap sch em a(R_2) = \varnothing$? Prodotto cartesiano.

RINOMINA

L'operatore di **rinomina** ci consente di cambiare il nome di un attributo in uno schema di una relazione. La rinomina è comoda quando vogliamo applicare un insieme di operazioni su differenti schemi, e quando vogliamo fare il join naturale di due relazioni su un insieme di attributi che sono comuni tra le due.

Sia r la relazione su uno schema di relazione R, A sia un attributo di $sch em a(R)$ e B un attributo non nello $sch em a(R)$. La rinomina, ρ, di A in B nella relazione r, è una relazione su uno schema di relazioni S, dove $sch em a(S) = (sch em a(R) \setminus \{A\}) \cup \{B\}$, definita da:

$$\rho(r, B = A) = \{t \mid \exists u \in r \,.\, t[sch em a(R) \setminus \{B\}] = u[sch em(R) \setminus \{A\}] \wedge t[B] = u[A]\}$$

È facilmente estendibile l'operatore di rinomina al caso di più coppie di rinomina:

$$\rho(r, B_1 = A_1, \ldots, B_n = A_n)$$

La funzione *rename*() implementa l'operatore di rinomina in dplyr:

```
planes
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type      manufacturer  model engines seats speed engine
##    <chr>   <int> <chr>     <chr>         <chr>   <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wi… EMBRAER       EMB-1…      2    55    NA Turbo…
##  2 N102UW   1998 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  3 N103US   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  4 N104UW   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  5 N10575   2002 Fixed wi… EMBRAER       EMB-1…      2    55    NA Turbo…
##  6 N105UW   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  7 N107US   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  8 N108UW   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
##  9 N109UW   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
## 10 N110UW   1999 Fixed wi… AIRBUS INDUS… A320-…      2   182    NA Turbo…
## # ... with 3,312 more rows
```

```
rename(planes, engine_type = engine, engine_number = engines)
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type       manufacturer  model engine_number seats speed
##    <chr>   <int> <chr>      <chr>         <chr>         <int> <int> <int>
##  1 N10156   2004 Fixed win… EMBRAER       EMB-1…            2    55    NA
##  2 N102UW   1998 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  3 N103US   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  4 N104UW   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  5 N10575   2002 Fixed win… EMBRAER       EMB-1…            2    55    NA
##  6 N105UW   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  7 N107US   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  8 N108UW   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
##  9 N109UW   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
## 10 N110UW   1999 Fixed win… AIRBUS INDUS… A320-…            2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine_type <chr>
```

È pregevole notare che il join può essere espresso tramite la selezione, rinomina, prodotto cartesiano e la proiezione.

Queste sono tutte e le sole operazioni presenti nell'algebra relazionale, nulla più.

Un **espressione di algebra relazionale** (o query) è una espressione ben definita composta di un numero finito di operatori dell'algebra relazionale i cui operandi sono gli schemi di relazione che possono essere trattati come variabili input della query. Una **risposta alla query** è ottenuta sostituendo tutti gli eventi/ occorrenze dello schema di relazione nella query con una relazione su uno schema è computando il risultato invocando gli operatori algebrici presenti nella query.

Per esempio, la seguente query recupera il mounth, day, origin, destination, carrier code e carrier name of flights che sia con valore "JetBlue Airways":

$$\pi(\sigma(flights \bowtie airlines, name = JetBlue Airways), \{month, day, origin, dest, carrier, name\})$$

Sarebbe comodo introdurre l'operatore pipe (>) nelle query dell'algebra relazionale. Per esempio, la precedente query potrebbe diventare più leggibile:

$$(flights \bowtie airlines) >$$
$$\sigma(name = JetBlue Airways) >$$
$$\pi(\{month, day, origin, dest, carrier, name\})$$

e risulta esser molto simile alla scrittura di dplyr:

```
inner_join(flights, airlines) %>% filter(name == "JetBlue Airways")
%>% select(month, day,  origin,  dest, carrier, name)
```

L'insieme della risposta della query è una relazione con 54,635 tuple e 6 attributi (mostriamo solo le prime dieci righe):

```
## # A tibble: 54,635 x 6
##    month   day origin dest  carrier name
##    <int> <int> <chr> <chr> <chr>   <chr>
##  1     1     1 JFK    BQN   B6      JetBlue Airways
##  2     1     1 EWR    FLL   B6      JetBlue Airways
##  3     1     1 JFK    MCO   B6      JetBlue Airways
##  4     1     1 JFK    PBI   B6      JetBlue Airways
##  5     1     1 JFK    TPA   B6      JetBlue Airways
##  6     1     1 JFK    BOS   B6      JetBlue Airways
##  7     1     1 LGA    FLL   B6      JetBlue Airways
##  8     1     1 EWR    PBI   B6      JetBlue Airways
##  9     1     1 JFK    RSW   B6      JetBlue Airways
## 10     1     1 JFK    SJU   B6      JetBlue Airways
## # ... with 54,625 more rows
```

Nota: la risposta di una espressione di albera relazionale è una relazione (insieme di tuple), quindi come un insieme non contiene duplicati. D'altra parte, il risultato dell'espressione in dplyr è un data frame. Mentre l'insieme di operazione su un data frame sempre rimuove i duplicati, le altre operazioni (in particolare le proiezioni) non rimuovono i duplicati. In questo quadro dplyr lavora come SQL. Un esempio è quello che abbiamo visto precedentemente.

PRACTICE

Write in both relational algebra and dplyr the following queries:

1. The plane tail number plane manufacturer of flights manufactured by EMBRAER. Mind that flights and planes tables share the year attribute with different semantics.

$$\sigma(planes, manufacturer = EMBRAER) > \pi(\{tailnum\})$$

```
filter(planes, manufacturer == "EMBRAER")  %>%
```

```
select(tailnum)
```

2. the flight number, month, day, hour, origin, and visibility of flights that took off with a visibility of 3 miles

$$(\text{flights} \bowtie \sigma(\rho(\text{planes}, \text{yearPlanes}=\text{year}), \text{manufacturer}=\text{EMBRAER})) > \pi(\{\text{flight}, \text{tailnum}, \text{manufacturer}\})$$

```
inner_join(flights, filter(planes, manufacturer == "EMBRAER"), by = "tailnum") %>%
  select(flight, tailnum, manufacturer )
```

3. flight paths of length 2 (X –> Y –> Z)

$$\rho(\pi(\text{flights}, \{\text{origin}, \text{dest}\}), \{X=\text{origin}, Y=\text{dest}\}) \bowtie \rho(\pi(\text{flights}, \{\text{origin}, \text{dest}\}), \{Y=\text{origin}, Z=\text{dest}\})$$

```
select(flights, X = origin, Y = dest) %>% inner_join(select(flights, Y = origin,
  Z = dest)) %>% distinct()
```

4. flight paths of length 3 (X –> Y –> W –> Z).

$$\rho(\pi(\text{flights}, \{\text{origin}, \text{dest}\}), \{X=\text{origin}, Y=\text{dest}\}) \bowtie \rho(\pi(\text{flights}, \{\text{origin}, \text{dest}\}), \{Y=\text{origin}, W=\text{dest}\}) \bowtie \rho(\pi(\text{flights}, \{\text{origin}, \text{dest}\}), \{W=\text{origin}, Z=\text{dest}\})$$

```
select(flights, X = origin, Y = dest) %>% inner_join(select(flights, Y = origin,
  W = dest)) %>% inner_join(select(flights, W = origin, Z = dest)) %>%
  distinct()
```

5. Can you generalize to paths of arbitrary length?

2.5.3 Calcolo dei Domini Relazionali

Se l'algebra relazione era un linguaggio procedurale la sua controparte **dichiarativa** è il **calcolo relazionale dei domini**. In cui dichiaro cosa voglio ma non come. Il calcolo relazionale è semplicemente una **logica a primo ordine**. Intuitivamente la logica di primo ordine ha a che fare con operatori booleani e usa anche i quantificatori $\forall$ e $\exists$. Per questi due connettivi esiste una relazione, nel senso che per ogni sarebbe il negato di esiste $\exists x \,.\, p(x) \equiv \neg \forall x \,.\, p(x)$.

Una logica di primo ordine è una logica in cui possiamo usare delle costanti, delle variabili, dei simboli di predicato (es. le relazioni, istanze di relazioni), delle funzioni (es. max, min, count, …) e poi si può combinare questo con gli operatori booleani e i $\forall$, $\exists$. Vedremo un parallelismo allora tra quantificatori algebra e logica.

In questo approccio logico un database relazione è interpretabile come un modello nella logica del primo ordine e una query è una formula di primo ordine nella medesima logica. La valutazione di una query su un DB corrisponde a controllare se il DB è un modello della formula di primo ordine rappresentato nella query (un problema conosciuto come **model checking**).

Model Checking, quindi, ci assicura che il nostro db sia una formula di primo ordine che corrisponde alla nostra interrogazione di primo ordine. Una espressione (o formula o query) del calcolo relazionale è di questa forma:

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_m)\}$$

Dove F è una formula ben definita, $x_1, \ldots, x_n$ sono le variabili libere presenti nella formula e $A_1, \ldots, A_n$ sono gli attributi, con $n \geq 0$. Quando $n = 0$, la formula è un'espressione Booleana. Noi assumeremo che tutti i simboli relazionali menzionati nella formula F ben definita stanno negli schemi relazionali che sono membro di un schema DB, cioè assumeremo di poter usare i simboli relazionali che stanno nella nostra base di dati. Vediamo di definire un po' per pezzi la sintassi e poi la semantica. La formula F la è definita ricorsivamente come segue. Una formula atomica è

1. $R(y_1, y_2, \ldots, y_n)$, dove R è un simbolo relazionale di grado n e tutti gli y_i sono o costanti o variabili
2. $X = Y$ e $X = c$, dove X, Y sono variabili e c è una costante

Le formule sono ora definite come segue:

1. Una formula atomica è una formula
2. Se F è una formula, allora lo sono anche $\neg F$ e (F)
3. Se F_1 e F_2 sono formule, allora lo sono anche $F_1 \vee F_2$ e $F_1 \wedge F_2$
4. Se F è una formula allora $\exists\, x : A\, .\, F$ è una formula, dove è una variabile e A è un attributo.

Noi scriviamo $F(x_1, x_2, \ldots, x_n)$ per una formula con variabili libere $x_1, x_2, \ldots, x_n$. Le variabili libere sono quelle che non sono sotto un segno di quantificatore. $\forall x\, .\, p(x, y, z)$, le variabili libere sono y e z. A questo punto si è voluto per cui dare un significato ad una espressione del calcolo, scriveremo allo $F(x_1, \ldots, x_n)$ una formula i cui argomenti sono variabili libere, ad esempio la formula q: $q(y, z) = \forall x\, .\, p(x, y, z)$. Sia $d = \{r_1, r_2, \ldots, r_m\}$ un DB di uno DB $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ e si consideri la query : $\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}$. Una tupla $\langle v_1, v_2, \ldots, v_n \rangle$ soddisfa la formula F se $v_i \in D(A_i)$, $\forall i$ e uno soddisfa una delle seguenti condizioni:

1. Se F è una formula atomica $R(y_1, y_2, \ldots, y_n)$, allora $R \in \mathcal{R}$ e la tupla t, risultante dalla sostituzione di v_i per ogni variabile y_i soddisfa $\in r$, dove è la relazione di R in d. Cioè la tupla soddisfa la relazione se soddisfa una istanza della relazione
2. Se F è una formula atomica $x_i = y_i$ allora $v_i = v$ è soddisfatta
3. Se F è una formula atomica $x_i = c$ allora $v_i = c$ è soddisfatta
4. Se F è una formula Booleana della forma $\neg F$, $F_1 \wedge F_2$ e $F_1 \vee F_2$, allora si applica la semantica del corrispondente connettivo logico.
5. .Se F è la formula $\exists\, x_i : A\, .\, G(x_1, x_2, \ldots, x_i, \ldots, x_n)$, allora $\langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \rangle$ soddisfa la formula F se esiste una costante $v_i \in D(A)$ tale che $\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$ soddisfi G.

Sembra una definizione complicata, ma se ci si pensa non lo è. A questo punto data una formula del calcolo relazionale $\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}$ il suo risultato rispetto dal datatase d è una relazione r con schema $\{A_1, \ldots, A_n\}$ contenente l'insieme di tutte le tuple che soddisfano F su d.

Ora presentiamo alcuni esempi di query del calcolo dei domini. Per ogni query, scriveremo prima l'espressione dell'Algeria relazione e poi la formula del calcolo relazionale.

Facciamo degli esempi per render il tutto più concreto.

SET OPERATORS

Gli operatori di insieme (intersezione, unione e differenza) sono implementati con connettivi Booleani (or, and, not) nel calcolo relazionale. Supponiamo che, lavorando sullo schema dei voli partiti da NY, selezioniamo un campione con la funzione sample_n, e proiettiamo queste relazioni sugli attributi month, day e flight, con il seguente codice in dplyr:

```
(flights1 = sample_n(flights, 10) %>% select(month, day, flight))
```

```
## # A tibble: 10 x 3
##    month   day flight
##    <int> <int>  <int>
##  1     7    16   4302
##  2     6    13     59
```

```
##  3     1    13   2167
##  4     2    20    369
##  5    10     7    483
##  6     7     1    574
##  7     1     7    487
##  8     6    26   1618
##  9     7    24   3662
## 10     5    20     41
```

E poi ne selezioniamo un altro:

```
(flights2 = sample_n(flights, 10) %>% select(month, day, flight))
```

```
## # A tibble: 10 x 3
##    month   day flight
##    <int> <int>  <int>
##  1     5    12   4376
##  2     7    26   1128
##  3     6    26   3341
##  4     5    25    202
##  5     3    27   1271
##  6     7    30    994
##  7     8    26     17
##  8     7    29   4193
##  9    11     5    301
## 10     6    18    731
```

A questo punto vogliamo trovare i voli che stanno o nel primo o nel secondo o in tutti e due (unione):

$$flights1 \cup flights2$$

$$\{x_1 : month, x_2 : day, x_3 : flight \mid flights1(x_1, x_2, x_3) \vee flights(x_1, x_2 x_3)\}$$

Stiamo usando la prima e la quarta delle regole di prima.
A questo punto vogliamo trovare l'intersezione tra i due, cioè i voli che stanno sia in flights1 che in flights2:

$$flights1 \cap flights2$$

$$\{x_1 : month, x_2 : day, x_3 : flight \mid flights1(x_1, x_2, x_3) \wedge flights(x_1, x_2 x_3)\}$$

Infine cerchiamo la differenza, cioè i voli che stanno in flights1 e non in flights 2:

$$flights1 \setminus flights2$$

$$\{x_1 : month, x_2 : day, x_3 : flight \mid flights1(x_1, x_2, x_3) \wedge \neg flights(x_1, x_2 x_3)\}$$

PROIEZIONE

Qua vediamo l'esistenziale. La proiezione, sarebbe come quantificare esistenzialmente nel calcolo relazionale. Per semplicità di lavoro lavoreremo su una relazione più piccola, cioè con meno attributi:

```
(flightsNarrow = flights %>% select(month, day, flight, tailnum, carrier))
```

```
## # A tibble: 336,776 x 5
```

```
##    month   day flight tailnum carrier
##    <int> <int>  <int> <chr>   <chr>
##  1     1     1   1545 N14228  UA
##  2     1     1   1714 N24211  UA
##  3     1     1   1141 N619AA  AA
```

```
##  4     1     1    725 N804JB  B6
##  5     1     1    461 N668DN  DL
##  6     1     1   1696 N39463  UA
##  7     1     1    507 N516JB  B6
##  8     1     1   5708 N829AS  EV
##  9     1     1     79 N593JB  B6
## 10     1     1    301 N3ALAA  AA
## # ... with 336,766 more rows
```

Vogliamo recuperare, cioè far la proiezione di month, day e flight. Nella tabelle precedente abbiamo rimpicciolito gli attributi per rendere la cosa più semplice da scrivere.

$$\pi(flightsNarrow, \{month, day, flight\})$$

$$\{x_1 : month, x_2 : day, x_3 : flight \mid \exists x_4 : tailnum . \exists x_5 : carrier . flightsNarrow(x_1, x_2, x_3, x_4, x_5)\}$$

Sto dicendo che voglio trovare i mesi, il giorno e i voli tali che esiste il tailnum e carrier e tali che la tupla appartiene alla relazione flightsNarrow.

SELEZIONE

La selezione in realtà è la più facile perché usa una formula logica. Recuperiamo i voli in flightsNarrow che partono a natale:

$$\sigma(flightsNarrow, day = 25 \wedge month = 12)$$

$$\{x_1 : month, x_2 : day, x_3 : flight, x_4 : tailnm, x_5 : carrier \mid flightsNarrow(x_1, x_2, x_3, x_4, x_5) \wedge (x_1 = 12) \wedge (x_2 = 25)\}$$

La prossima opzione non è esattamente la stessa cosa, perché la prima restituisce 5 attributi e la prossima solo 3:

$$\{x_3 : flight, x_4 : tailnm, x_5 : carrier \mid flightsNarrow(12,25,x_3, x_4, x_5)\}$$

JOIN

Il join non dovrebbe introdurre nulla di nuovo, perché abbiamo già introdotto tutto. Facciamo un join tra flightsNarrow e la relazione Airlines e prendiamo i voli e le corrispondenti compagnie aeree:

$$flightsNarrow \bowtie airlines$$

$$\{x_1 : month, x_2 : day, x_3 : flight, x_4 : tailnum, x_5 : carrier, x_6 : name \mid flightNarrow(x_1, x_2, x_3, x_4, x_5) \wedge airlines(x_5, x_6)\}$$

Scrivendo x_5 in diversi punti forzo a far si che siano uguali.

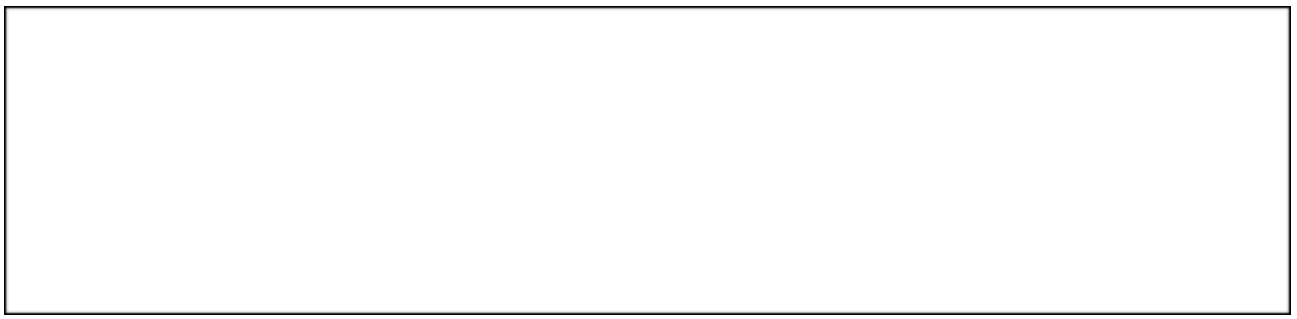
PRACTICE

Consider the (narrowed) relations:

```
(flightsNarrow =  flights %>%  select(flight, tailnum, origin, dest))
```

```
## # A tibble: 336,776 x 4
##    flight tailnum origin dest
##     <int> <chr>   <chr>  <chr>
##  1   1545 N14228  EWR    IAH
##  2   1714 N24211  LGA    IAH
```

```
##  3   1141 N619AA  JFK    MIA
##  4    725 N804JB  JFK    BQN
##  5    461 N668DN  LGA    ATL
##  6   1696 N39463  EWR    ORD
##  7    507 N516JB  EWR    FLL
```



```
##  8   5708 N829AS  LGA    IAD
##  9     79 N593JB  JFK    MCO
## 10    301 N3ALAA  LGA    ORD
## # ... with 336,766 more rows
```

```
(planesNarrow = planes %>% select(tailnum, manufacturer))
```

```
## # A tibble: 3,322 x 2
##    tailnum manufacturer
##    <chr>   <chr>
##  1 N10156  EMBRAER
##  2 N102UW  AIRBUS INDUSTRIE
##  3 N103US  AIRBUS INDUSTRIE
##  4 N104UW  AIRBUS INDUSTRIE
##  5 N10575  EMBRAER
##  6 N105UW  AIRBUS INDUSTRIE
##  7 N107US  AIRBUS INDUSTRIE
##  8 N108UW  AIRBUS INDUSTRIE
##  9 N109UW  AIRBUS INDUSTRIE
## 10 N110UW  AIRBUS INDUSTRIE
## # ... with 3,312 more rows
```

Write the following query in relational calculus: Find the flight number, the plane tail number, and plane manufacturer of flights manufactured by EMBRAER.

$$\{x_1 : flight, x_2 : tailnum, x_3 : manufacturer \mid \exists\, x_4 : origin\ .\ \exists\, x_5 : dest\ .\ flightsNarrow(x_1, x_2, x_4, x_5) \wedge planesNarrow(x_2, x_3) \wedge x_3 = 'EMBRAER'\}$$

Consider the (narrowed) relation:

```
(flightsNarrow = flights %>% select(origin, dest))
```

```
## # A tibble: 336,776 x 2
##    origin dest
##    <chr>  <chr>
##  1 EWR    IAH
##  2 LGA    IAH
##  3 JFK    MIA
##  4 JFK    BQN
##  5 LGA    ATL
##  6 EWR    ORD
##  7 EWR    FLL
##  8 LGA    IAD
##  9 JFK    MCO
## 10 LGA    ORD
## # ... with 336,766 more rows
```

Find the flight paths of length 2 (X –> Y –> Z).

$$\{x : origin, y : dest, z : dest \mid flightsNarrow(x, y) \wedge flightsNarrow(y, z)\}$$

Find the flight paths of length 3 (X –> Y –> W –> Z).

$$\{x : origin, y : dest, z : dest, w : dest \mid flightsNarrow(x, y) \wedge flightsNarrow(y, z) \wedge flightsNarrow(z, w)\}$$

EQUIVALENZA

Vogliamo accennare alla dimostrazione che ci portiamo da un po' di tempo per la quale la **chiusura transitiva** non è esprimibile in SQL, nell'algebra relazionale e nel calcolo dei domini relazionali.

Innanzitutto precisiamo che l'insieme risultato da una espressione di una algebra relazionale è sempre un insieme finito. Non è così per il calcolo, partendo da insiemi finiti possiamo passare ad insiemi infiniti, per esempio. Cos'è una query del calcolo relazionale con un insieme di risposta infinito?

Sia P uno schema di relazione con un attributo A e Q uno schema con un solo attributo B, entrambi gli attributi sono definiti su un dominio dei numeri naturali. Sia d un DB che contiene una relazione non vuota p per lo schema P e una relazione q per Q. Ricordando che A ha come dominio i numeri naturali e nella nostra relazione vanno, ad esempio da 1 a 10 il risultato di $\{x$ da l'insieme costituito da numeri $: A \mid \neg P(x)\}$

maggiori di dieci e lo zero, quindi infinito. Mentre $\{x : a , y : B \mid P(x) \wedge Q(y)\}$, ammettendo che P contenga i numeri 0,1e 2 abbiamo un risultato infinito. Dobbiamo allora fare delle restrizioni sintattiche in modo da ottenere solo formule che danno risultati finiti. Questo è possibile, cioè possiamo limitare i casi in casi finiti, ma noi non vedremo la modalità

Assunto questo l'algebra relazione e il calcolo relazionale consentito hanno lo stesso **potere espressivo**, cioè sono equivalenti. Cioè data una espressione dell'algebra possiamo trovare una formula della logica e viceversa. A questo punto possiamo trovare delle formule che non stanno in questo linguaggio per esempio quella che abbiamo visto dettata dalla transitività arbitraria dei voli. Intuitivamente, trovando un percorso di lunghezza $k + 1$ richiede un query di k join, quindi quando il numero di passi non è delimitato anche la query risulterà essere non delimitata. Però, una query è un oggetto finito.

Mostriamo che una chiusura transitiva non è definibile nella logica di primo ordine. A questo scopo, useremo il *Compactess Theorem* di primo ordine, che dichiara che: un insieme di proposizioni di primo ordine ha un modello se e solo se tutti i sottoinsiemi finiti hanno un modello.

Data una relazione binaria R la sua **chiusura transitiva** R^* è la più piccola relazione tale che:

1. $s e x R y a l l or a x R^* y$;
2. $s e x R w e x R^* y a l l or a x R^* y$.

Data un simulo di relazione R e R^* , supponiamo che ci sia una formula di primo ordine $\phi(R , R^*)$ che esprime che R^* è la chiusura transitiva di R. Si consideri inoltre il simbolo costante $c_1 \neq c_2$ e, $\forall n \geq 0$, si definisce:

$$\pi_n = \neg(\exists x_1 . \exists x_2 \ldots \exists x_2 . c_1 R x_1 \wedge x_1 R x_2 \wedge \ldots \wedge x_n R c_2)$$

La formula π_n dice che non ci sono $R-$ percorsi, cioè un cammino su R di lunghezza $n + 1$ che connettono c_1 e c_2 quindi la congiunzione di tutte le formule π_n per $n \geq 0$ dice che non ci sono percorsi di lunghezza arbitraria tra c_1 e c_2. Si costruiva la teoria infinita:

$$\Psi = \{\phi(R , R^*), c_1 R^* c_2\} \cup \{\pi_i\}_{i \geq 0}$$

La teoria Ψ è inconsistente, così è, non ha un modello, perché $c_1 R^* c_2$ implica che ci sia un cammino di R di arbitraria lunghezza che connette c_1 e c_2, ma la formula π_i per $i \geq 0$ vieta l'esistenza di questo percorso. Tuttavia si noti che ogni sottoinsieme finito della teoria Ψ è consistente. Dal Compacyeness Theorem,

abbiamo una contradizione. Allora $\phi(R , R^*)$non può esistere nella logica del primo ordine.

La teoria è un insieme di formule in questo caso, un modello è una istanza delle base di dati, cioè bisogno dare un senso a quel predicato. (R-path = cammino su R, $x R y$ = "andare da x a y")

Sfortunatamente, non abbiamo finito. Il teorema di compattezza non tiene conto dei modelli finiti, che è il caso dei DB. Questo allora è un contro esempio: sia λ_n una formula ce ice check sono almeno distinti elementi nel dominio. Questo può essere espresso nella prima logica, per esempio:

$$\lambda_3 = \exists x_1 . \exists x_2 . \exists X_3 . (x_1 \neq x_2) \wedge (x_2 \neq x_3) \wedge (x_3 \neq x_1)$$

Consideriamo la teoria:

$$\Psi = \{\lambda_n \mid n > 0\}$$

Ogni sottoinsieme finito di Ψ ha un modello finito, ma Ψ non ha un modello finito (ma solo infiniti).

2.7 Normalizzazione

Inizieremo con un esempio. La **teoria della normalizzazione** stabilisce dei criteri se il modello logico relazionale soddisfa o meno determinati criteri definiti in maniera rigorosa. Vogliamo gestire una base di dati per gestire gli esami orali di una università, tramite la relazione:

$$R = \{Student, Course, Chapter, Time, Room\}.$$

Ogni capitolo è il titolo o il numero di un capitolo dal libro del testo di un corso. Vincoli:
- Ogni studente può esser interrogato su più di un capitolo e i capitoli sono diversi per ogni studente.
- Ogni studente fa un solo esame per corso.
- Ogni studente fa l'esame in una sola aula

Una possibilità è scrivere un solo schema relazionale come prima. $R = \{Student, Course, Chapter, Time, Room\}$. Non è la miglior idea perché abbiamo un problema di ridondanza di informazione e quindi ci saranno problemi di anomalie di modificazione dei dati. Dal punto di vista della teoria della formalizzazione si possono scrivere i vincoli in maniera più formale tramite la **dipendenza funzionale.** Una dipendenza funzionale $X -> Y$, dove X e Y sono attributi, esprime un vincolo di integrità sugli attributi. A una dipendenza funzionale solitamente è associato uno schema relazionale. Ciò implica che ogni qualvolta che due tuple coincidono su valori in X devono coincidere anche su Y.

R schema relazionale d.f. $X -> Y$
r istanza su R
r soddisfa la d.f. se $\forall t, t' \in r, t[X] = t'[X] -> t[Y] = t'[Y]$

Esempio: R(A), $\varnothing -> A$, le istanze valide sullo schema R, solo due istanze sono valide: quella vuota oppure una relazione con una sola tupla. Generalmente una dipendenza di questo tipo forza le assunzioni di valori del secondo attributo.

Le dipendenze funzionali servono a catturare determinati vincoli di integrità. Le dipendenze sono date solitamente a priori insieme allo schema. Nel nostro esempio, tramite l'**analisi dei requisiti,** troviamo le dipendenze che vengono fuori sono:

$Student, Time -> Room$
$Student, Course -> Time, Room$
$Time, Room -> Course$

Quando parliamo di dipendenze funzionale si può parlare di **conseguenza**, cioè:

$\mathcal{F} \models X -> Y, \mathcal{F} implica X -> Y$ Cioè ogni volta che soddisfo $\mathcal{F}$ soddisfa anche $X -> Y$.

Esempio: $F \models \{A -> B, B -> C\}$ allora $F \models A -> C$, proprietà transitiva. Per brevità si può scrivere così:

$$\frac{A -> B \quad B -> C}{A -> C}$$

Ci sono altre conseguenze logiche.

$$\frac{X -> YZ}{X -> Y}$$ dove $YZ = Y \cup Z$

$$\frac{X -> A_1, \ldots, A_k}{X -> A_1 \ldots X -> A_k}$$ posso togliere attributi sul lato destro

$X -> Y$ posso aggiungere attributi sul lato sinistro

$X \cancel{Y -> YX} -> Y$ Riflessività. Un insieme di attributi determina sempre un suo sottoinsieme

Regole dell'Aumento

$XZ -> YZ$

In termini di dipendenza funzionali il concetto di chiave può esser reso rigoroso tramite due definizioni. Una **super chiave** è un insieme di attributi che determina univocamente ogni istanza: X è una super chiave se e solo se $F \vdash X -> attr(R)$.

Una chiave non è altro che una super chiave minimale. Quindi le dipendenze funzionali ci possono anche aiutare nella definizione di una chiave.

Chiusura di un insieme di attributi di X rispetto a *F:* è l'insieme di tutti gli attributi A t.c. $F \vdash X -> A$. Cioè insieme degli attributi che sono determinati funzionalmente da *X*.

Teorema: $F \vdash X \rightarrow Y$ se e solo se $Y \subseteq X^+_F$

Questo è un risultato potete, perché questo risultato ci consente di trasformare una condizione di tipo semantico in una condizione algoritmica. Vediamo degli esempi per meglio capire:

Sia F = $\{AB \rightarrow CD, C \rightarrow E, B \rightarrow A\}$. Calcoliamo la chiusura di AB rispetto a F. Poniamo $X(0) = AB$. Per calcolare $X(1)$ cerchiamo in F le dipendenze il cui lato sinistro sia A, B oppure AB: otteniamo $AB \rightarrow CD$ e $B \rightarrow A$. Perciò aggiungiamo a $X(1)$ gli attributi che compongono i lati destri di tali dipendenze: C, D e A. Si ha $X(1) = X(0) \cup \{C, D, A\} = ABCD$. Per calcolare $X(2)$ cerchiamo le dipendenze il cui lato sinistro è contenuto in $X(1)$: $AB \rightarrow CD$, $C \rightarrow E$, $B \rightarrow A$. Otteniamo: $X(2) = X(1) \cup \{C, D, E, A\} = ABCDE$. Essendo $X(2)$ l'insieme di tutti gli attributi, sicuramente $X(3) = X(2)$, e quindi abbiamo finito.

Per trovare le chiavi ci sono alcune regole interessanti:

- Se un attributo compare solo a destra delle relazioni funzionali non può far parte di nessuna chiave, nell'esempio l'attributo E.
- Gli attributi che stanno solo nei lati sinistri devono far assolutamente parte di tutte le chiavi delle relazioni (nell'esempio l'attributo B, nel nostro esempi ogni chiave deve contenere B, nel nostro caso solo B è chiave).

Tornando all'esempio iniziale: la dipendenza $Student(S), Course(C) -> Room(R)$ è

$SC^+ = SCTR$ è una conseguenza logica

Allora è conseguenza logica delle altre dipendenze. Tutti gli attributi compaiono sia sul lato destro che sinistro tranne S e H, allora tutte le chiavi devono contenere sia S che H (Chapter). Però se calcolo la chiusura di SH questa risulta essere solo SH. Allora possiamo procedere per tentativi: tutte le chiavi devono contenere SH, ma non solo SH. Quindi le chiavi devono avere almeno tre attributi. Allora possiamo fare le chiusure con le triple:

$SHC^+ = SHCTR$ è chiave

$SHT^+ = SHTRC$ è chiave

$SHR^+ = SHR$ Questa non è chiave

Allora abbiamo due possibili chiavi. Il nostro problema è che il nostro schema così com'è non ci piace, perché abbiamo delle ridondanze. Per risolvere il problema decomponiamo lo schema, cioè facciamo delle proiezioni per costruire due o più schemi, che ha come contenuti, da un aspetto informativo, medesimi di prima. Desideriamo che siano soddisfatte due proprietà:

1. Se proiettiamo voglio essere in grado di ricostruire l'istanza iniziale con una procedura di join. $r = \bowtie^{K}_{i=1} \pi_{R_i}(r)$, dove $R_1, \ldots, R_k$ è decomposizione di R. Questa proprietà è detta **lossless join** (senza perdita di informazione:

Esempio:
$R(A, B, C, D)$
$\mathcal{F} = \{A -> B, C -> D\}$
Creiamo due relazioni $R_1(A, B)$ e $R_2(C, D)$, e non risultano essere una decomposizione possibile perché con un contro esempio:

R

A	B	C	D
1	a	10	x
2	b	20	Y

R1

A	B
1	a
2	b

R2

C	D
10	x
20	Y

R1 \Join R2

A	B	C	D
1	a	10	x
2	b	20	Y
1	a	20	Y
2	b	10	x

Che risulta essere diverso da R

2. La decomposizione deve preservare le dipendenze. Cioè vogliamo che $\cup^{k}_{i=1} \mathcal{F}_i \equiv \mathcal{F}, (\forall X -> Y \in \cup \mathcal{F}_i . \mathcal{F} \vdash X -> Y \; e \; \forall X -> Y \in \mathcal{F}. \cup \mathcal{F}_i \vdash X -> Y)$. Allora si dice che è **fedele** o preserva le dipendenze.

Esempio: $R(A, B, C)$, $\mathcal{F} = \{A -> B, B -> C, C -> A\}$ da cui scomponiamo due relazioni $R_1(A, B)$ e $R_2(B, C)$. La dipendenza $c -> A$ può non essere associata/proiettata ne su R_1 né su R_2 perché in R_1 non serve C e in R_2 si associa A, grazie alla proprietà transitiva. Tuttavia questa dipendenza è preservata grazie alla transizione, poiché $C -> A$ e allora se $B -> C$ allora $B -> A$, similmente $C -> B$.

Anche queste dipendenze bisogna considerare, quindi non solo quelle scritte esplicitamente, ma anche quelle implicate. Allora se prendo l'unione di questi insiemi /schemi abbiamo che $C -> A$, anche se non è proiettata in modo esplicito. Questo composizione quindi risulta fedele, oltre ad essere Lossless.

A partire da queste nozioni possiamo arrivare a diverse definizioni di **forme normali**, noi daremo solo una che è chiamata **forma normale di Boyce-Codd Normal Form**:

Definizione: uno schema (R, F) è in **forma normale di Boyce- Codd** (abbreviato **BCNF**) se per ogni dipendenza non banale $X \rightarrow A \in$ F si ha che X è superchiave di R. Una decomposizione ϱ è in BCNF se ogni schema di ϱ è in BCNF.

Questa definizione ci porta a dire che nell'esempio iniziale il problema è che se ho un esame che viene fatto da dieci studenti avremo l'informazione che l'appello è stato in data e in luogo viene ripetuta più e più volte. Questa definizione vuole eliminare questa ridondanza, esempio:

R .

S	C	H	T	R
S1	C1		T1	
S1	C1		?	

La casella in posizione (2,4) è ignota, ma sappiamo il suo valore perché deve essere uguale a T1. Questo vuol dire che c'è ridondanza, perché posso trovare il valore cercando una tupla e usare una dipendenza relazionale.

Allora vogliamo creare due schemi che scompongono in due schemi che siano BCNF. C'è un algoritmo molto bello che fa questo. Si prende una dipendenza ($SC -> T$) e creiamo uno schema con questi poi prendiamo uno schema composto dagli atri attributi $R_2(H, R, T)$.

Le decomposizioni BCNF sono sempre lossless, ma possono perdere delle dipendenze.

Un altro problema è quello di attribuire un significato a questi schemi creati, ad esempio lo schema creato $R_1(S, C, T)$, lo studente ha superato un certo esame in una certa data. Mentre R_2 non è una tabella naturale, nel senso che creeremmo naturalmente per significato. Questo vuol dire che la teoria della normalizzazione da garanzie su un aspetto matematico e teorico e non su un aspetto interpretativo.

Fondamentalmente abbiamo ora due metodi per creare una base di dati, uno è il modello relazionale e uno è la normalizzazione. Questi son due approcci complementari che possono esser utilizzati per aiutare l'altro. Il primo è creato da noi, che magari però contiene la ridondanza, il secondo invece non crea ridondanza però gli schemi creati posso esser poco interpretabili. Quindi il consiglio è quello di creare una progettazione relazionale con la nostra testa, poi però si cerchi di, dopo aver creato il modello relazionale dopo quello concettuale, analizzare se gli schemi sono in una forma normale. Se facessimo una relazione con gli attributi $flightnum, origin, destination, plan$ allora se sappiamo plane allora sappiamo $e, manufacturer$

anche manufacturer. Quindi costruito questo schema potremmo creare un'altra relazione proiettando questi attributi ad esempio ($flightnum, origin, destination, plan$ e ($plane, manufacturer$) dove e)

abbiamo una chiave per tabella e una chiave esterna. Quindi il consiglio è quello di utilizzare questa strategia mista.

Gestione della controrenza e gestione delle transizioni sono due topic che abbiamo saltato, ma molto interessanti, se si vuole leggere dai libri suggeriti (LL99 e EN16)

3. Data Science

In questa parte usare fondamentalmente due software: R e RStudio. Inizieremo con un breve tour di r.

3.1 Inside R

Il risultato di 1+1 su RStudio è:

```
1+1
```

```
[1] 2
```

Quel uno iniziale ci indica che anche il risultato è un vettore. R lavora in termini vettoriali. Vediamo ora operatori basilari, ricordiamo che la divisione intera si fa con il doppio percentuale %%:

```
# arithmetic
1 + 2 * 4 - 2 / 2

# integer division
31 %/% 3

# modulus
31 %% 3

# exponents
2^10

# comparison, si ricorda che il confronto è con il doppio uguale ==, l'uguale è
# assegnazione
1 == 1
1 != 1
1 < 1
1 <= 1

# logic operators
# conjunction
TRUE & TRUE
FALSE & TRUE
TRUE & FALSE
FALSE &
FALSE

# disjunction
TRUE | TRUE
FALSE | TRUE
TRUE | FALSE
FALSE | FALSE

# negation
!TRUE
!FALSE

# exclusive disjunction
xor(TRUE, TRUE) xor(TRUE,
FALSE) xor(FALSE, TRUE)
xor(FALSE, FALSE)
```

Per vedere i risultati copiare i comandi su R. Possiamo usare gli operatori appena visti o la versione più lunga, la quale è più efficiente, perché valuta da sinistra a destra e appena il risultato è chiaro da il risultato, la forma corta valuta tutto. L'and lungo sarebbe && e l'or lungo è ||. Questo vuol dire che se scriviamo :

```
TRUE || ESP
```

Quindi un valore con valore logico vero o una espressione, questo comando da TRUE senza analizzare l'espressione, cosa che avrebbe fatto con |.

There are a few **special values**:

- the value `NA` (not available) is used to represent missing values;
- the value `NULL` is the null object (not to be confused with NULL in databases);
- the value `Inf` stands for positive infinity;
- the value `NaN` (not a number) is the result of a computation that makes no sense.

```
NA & TRUE
NA &
FALSE NA |
TRUE NA |
FALSE
!NA

2^1024
1/0
0 / 0
Inf - Inf
```

```
[1] NA
[1] FALSE
[1] TRUE
[1] NA
[1] NA
[1] Inf
[1] Inf
[1] NaN
[1] NaN
```

Ci sono tre modi per fare assegnazioni alle variabili:

```
x = 45
x <- 45
45 -> x
```

Il modo consigliato è il terzo.
Per vedere il valore della variabile, basti:

```
x
# or
print(x)
# or print the structure of the object str(x)
```

Non ci sono tanti algoritmi, ma ci sono tante strutture di dati e ne abbiamo sostanzialmente 5:

- Il vettore
- La matrice
- La lista
- L'array
- Data frame

```
Dim Homogeneous      Heterogeneous
1d  Atomic vector    List
2d  Matrix           Data frame
nd  Array
```

Eterogenea vuol dire che può contenere tipi diversi. Se abbiamo un oggetto x per sapere che tipo sia:

```
typeof(x)
```

Questo è un duble, da come lo abbiamo costruito prima.

3.1.1 Vettori

Un vettore è una sequenza di elementi che hanno tuti lo stesso tipo, e si usa le funzione c() o la funzione seq. Se volessi un vettore di interi devo scrivere i numeri seguiti da L:

```
y = c(1L, 2L, 3L)
typeof(y)
```

Esiste la somma tra vettori. La somma ed altri operatori come il prodotto e la divisione è elemento per elemento del vettore e non nel complesso.

```
# element-wise sum
c(1, 2, 3, 4) + c(10, 20, 30, 40)

# element-wise product
c(1, 2, 3, 4) * c(10, 20, 30, 40)

# scalar product
c(1, 2, 3, 4) %*% c(10, 20, 30, 40)
```

Bisogna stare attenti al **recicling:**

```
c(1, 2, 3, 4) + 10
c(1, 2, 3, 4) + c(10, 20)
```

I vettori possono essere anche di stringhe o di booleani.

```
a = 1:10 a
a[5]
a[c(1, 5, 10)]
# Metto il meno per togliere l'elemento a[-1]
a[-c(1, 5, 10)]
# Seleziona solo gli elementi che sono maggiori di 5. a > 5
# Per prendere solo gli elementi che del vettore hanno valore maggiore di 5 a[a > 5]
```

Tutti gli elementi di un vettore atomico devono essere dallo steso tipo, allora quando cerchiamo di combinare differenti tipi saranno tradotti nel tipo più flessibile (coercizione). Dal meno flessibile al più flessibile:logical, integer, double, and character.

```
as.integer(FALSE)
as.integer(TRUE)
x = c(TRUE, TRUE, FALSE, FALSE)
```

```
sum(x) mean(x)
as.logical(0)
as.logical(1)

as.double(0L)
as.integer(0.5)

as.character(0.5)
as.double("0.5")
as.numeric(“a")
```

Si usa il comando name per assegnare un nome ai valori di un vettore. Il **fattore** è un vettore che rappresenta una variabile categoriale (colore degli occhi, sesso, ecc.), cioè un vettore di interi che ha un attributo *levels* che mi dice quali sono i valori ammessi.

```
x = factor(c("male", "female", "female", "male", "male"))
x typeof(x)
levels(x)
table(x)

# You can't use values that are not levels x[1] = "unknown"
x
```

```
[1] male   female female male   male
Levels: female male
[1] "integer"
[1] "female" "male"
x
```

È interessante fare table() per notare le frequenze per livello. Se assegnamo il valore “unknown” genero un elemento di tipo NA, come sopra. Attenzione che storicamente proprio perché nato dagli statistici, tutti i vettori di stringhe vengono trasformati in factors, questo non è bene perché è bene creare factors solo quando conosciamo in anticipo quali sono i valori noti. Quindi se stiamo importando un DSet allora se abbiamo una colonna che è di stringhe viene convertita in factor, possiamo eliminare questa cosa con stringeAsFactors=FALSE.

3.1.2 LISTE

Arriviamo quindi ad un argomento spinoso: le **liste**. Sono molto importanti perché si usano parecchio, vedremo che un DFrame non è altro che una lista. Abbiamo una sola dimensione, può essere ricorsiva (contenere altre liste o in generale uno strumento strutturale come un vettore) può contenere valori di diverso tipo. Questo ci permette di creare liste che contengono liste, che contengono liste, ecc.

```
l = list(thing = "hat", size = 8.25, female = "TRUE")
l

# an element
l$thing l[[1]]
l[["thing"]]

# a sublist l[c(1,
2)]
```

```
l[c("thing", “size”)]
```

Come vediamo per estrarre il primo elemento della lista abbiamo tre metodi, se vogliamo estrarre un sottolista dobbiamo usare il doppio indice. Nota: l[1] abbiamo la sottolista che contiene il primo elemento e l[[1]] è il primo elemento. Bisogna differenziare il contenitore dal contenuto.

```
l[1]
typeof(l[1])

l[[1]]
typeof(l[[1]])
```

Se x è un treno fatto da più vagoni allora x[[5]] è la carrozza 5 mentre x[4:6] è il treno della carrozza 4 a 6.
Per rimuovere gli elementi da uno lista, anzitutto per aggiungere l’elemento basta che lo nomini. Se si vuole eliminare un elemento o si deve assegnare a NULL, cioè all’oggetto vuoto.

```
l = list(a = 1, b = 2)
l$c = 3 l
l$c = NULL
l
```

Una lista può contenere un vettore.

```
l = list(thing = "hat", prices = c(8.25, 10.5), female = "TRUE")
l l$prices[1]
```

Con l’ultimo comando prendiamo il primo elemento del vettore contenuto nella lista. Oppure potrebbe essere ricorsiva, quindi contenere una lista al suo interno.

```
l = list(1, list(1, 2, 3), list("a", 1, list("TRUE", “FALSE")))
# Troviamo la lista list(1,2,3)
l[[2]]
# troviamo l’elemento 1 della lista list(1,2,3)
l[[2]][[1]]
# troviamo l’elemento TRUE della lista list(TRUE, FALSE)
list[[3]][[3]][[1]]
```

Si possono combinare le liste con:

```
a = list(1, 2, 3) b =
list(3, 2, 1) c(a, b)
```

3.1.3 MATRICI

Non le useremo troppo, comunque è bene sapere che ci sono. È una struttura di dati omogenea però ha righe è colonne, cioè è bidimensionale:

```
M = matrix(data = 1:9, nrow = 3, ncol = 3, byrow = TRUE) M
N = matrix(data = 1:9, nrow = 3, ncol = 3) N

nrow(M)
ncol(M)
dim(M)
```

Bisogna anche specificare se leggere per righe e per colonne, di default è per colonna. Anche la matrice non è nient'altro che un vettore con l'attributo dim:

```
x = 1:9 x
dim(x) = c(3, 3)
x
```

Per accedere agli elementi si possono usare gli indici (in questo caso sono due):

```
# element
M[1, 2]

# frist row
M[1, ]

# first column
M[ ,1]

# sub-matrix
M[1:2, 1:2] M[-3,
-3]

# diagonal
diag(M)
diag(M) = 0
M
```

Le operazioni su matrici avvengono elemento per elemento, non nel complesso come per i vettori:

```
# element-wise sum
M + N

# element-wise product
M * N

# matrix product
M %*% N

# matrix transpose t(M)

# matrix inverse
C = matrix(c(1,0,1, 1,1,1, 1,1,0), nrow=3, ncol=3, byrow=TRUE) D = solve(C)
D
D %*%
C C %*%
D

# linear systems C x = b
C
b = c(2,1,3)
# the system is:
# x1 + x3 = 2
# x1 + x2 + x3 = 1
# x1 + x2 = 3 x
= solve(C,b)
x
C %*% x
```

```
# matrix spectrum, cioè l'insieme degli autovalori e autovettori spectrum = eigen(C)
spectrum$vectors
spectrum$values

spectrum2 = eigen(t(C))
spectrum2$vectors
spectrum2$values
```

Possiamo aggiungere righe e colonne:

```
M
rbind(M, 10:12)
# this makes a copy of M M
# modify M with
M = rbind(M, 10:12) M

M = cbind(M, seq(4, 16, 4)) M
```

Possiamo dare dei nomi sia alle righe che alle colonne:

```
rownames(M) = letters[1:nrow(M)]
colnames(M) = LETTERS[1:ncol(M)] M
M["a", "A"]
M["a", ]
M[ ,"A"]
```

3.1.4 ARRAY

Essi sono matrici multi dimensionali, ad esempio:

```
A = array(1:27, dim=c(3, 3, 3)) A
A[ , , 1] A[ ,
1, 1] A[1 , ,
1] A[1, 1, 1]
```

3.1.5 DATA FRAME

È una lista di vettori chiamati colonne che però rispetto alla lista devono avere tutti la stessa dimensione, ottengo così una struttura rettangolare. È qualcosa di molto vicino alla tabella di un DB relazionale. Ovviamente essendo una lista le colone possono avere tipi diversi, all'interno della stessa colonna ci deve essere dello stesso tipo. Esempio:

```
team = c("Inter", "Milan", "Roma", "Palermo")
score = c(59, 58, 53, 46) win =
c(17, 17, 15, 13) tie = c(8, 7, 8, 7)
lost = c(3, 4, 5, 8)
# Notare che si usa stringsAsFactors = FALSE per le motivazioni sopraccitate league = data.frame(team,
score, win, tie, lost, stringsAsFactors = FALSE) league
```

Possiamo prendere solo alcuni elementi:

```
#  first row
league[1, ]

#  first column league[
,1] league[ ,"team"]

league[1:2, 1:2]
league[1:2, c("team", “score")]
```

Le colonne hanno immediatamente un nome, cioè delle variabili che passo. Si può accedere alle colonne anche tramite il loro nome. Possiamo aggiungere le colonne e le righe, quest’ultimo è più complicato, ma basta fare attenzione:

```
rbind(league,  data.frame(team =  "Lazio", score =  44, win   =  12, tie =  8, lost =  8))
cbind(league, goals =  c(45, 43, 38, 36))
```

I nomi della nuova tuple devono essere gli stessi del DFrame. Un DFrame è una lisa e quindi posso accedere agli elementi con gli accessi tipici delle liste. Se tolgo le doppia parentesi e ne ho solo una allora abbiamo un DFrame, se no abbiamo l’elemento cercato del suo tipo.

```
#  a data frame is  a list
typeof(league) league$team
league[[1]]
league[league$team ==  "Inter", ]
league[league$score ==  max(league$score), ]

nrow(league)
ncol(league)
rownames(league)
colnames(league)
```

Attenzione: dato che un DFrame è una lista di vettori e un vettore può essere una lista, possiamo creare due DFrame con delle colonne di tipo lista, questo fa si che gli elementi di queste colonne liste possano essere complicate finché vogliamo, quindi potrebbero essere anche dei DFrame, quindi possiamo avere strutture ricorsive o annidate, cosa che non può succedere nel modello relazionale perché gli elementi sono atomici.

```
#  a data  frame with a list column df =
data.frame(
    x  =  I(list(1:3, 4:6)),
    y  =  c("On the  left  there is a list", "The  same here!"), stringsAsFactors =
    FALSE
)
#  Questo è un   DFrame   che contiene delle strutture  non   atomiche (vettori)
str(df)

#  a data  frame with data  frame elements,  ha la  prima colonna che contiene  una lista con
#  dentro il DFrame   df. È  un   DFrame   che contiene  un   DFrame df2 =
data.frame(
    x  =  I(list(df, df)),
    y  =  c("On the  left  there is a list", "The  same here!"), stringsAsFactors =
    FALSE
)
str(df2)
```

Questo lo utilizzeremo soprattutto nel caso di regressione lineare. Sapendo che in R una regressione lineare è un DFrame, lo immetteremo nel DFrame di interesse.

3.1.6 CONDIZIONALE E RIPETIZIONE

R è un linguaggio orientato all'oggetto. Quindi è possibile scrivere dei cicli if, while e for. Bisogna usarli con parsimonia.

```
x = 49
if (x %% 7 == 0) x else -x


x = 108 i
= 2
while (i <= x/2) {
  if (x %% i == 0) print(i)
  i = i + 1;
}


for (i in 2:(x/2)) {
    if (x %% i == 0) print(i)
}
```

3.1.7 FUNZIONI

Possiamo scrivere delle funzioni con r. Molte funzioni di R, anzi tutte, hanno dei valori prefissati, che possiamo andare a modificare, per questo R risulta molto flessibile, come la funzione log:

```
log args(log)
log(x = 128, base = 2)
log(128, 2)
log(128)
```

O possiamo creare funzioni:

```
euclidean = function(x=0, y=0) {sqrt(x^2 + y^2)}


euclidean(1, 1)
euclidean(1)
euclidean()
```

Possiamo scrivere delle funzioni ricorsive come il fattoriale:

```
factorial = function(x) {
  if (x == 0) 1 else x * factorial(x-1)
}
factorial(5)
```

Possiamo scrivere anche dei funzionali, cioè delle funzioni che hanno come parametro delle funzioni

```
g = function(f, n) {
  sum = 0
  for (i in 0:n) sum = sum + f(i)
  return(sum)
}


g(factorial, 5)
```

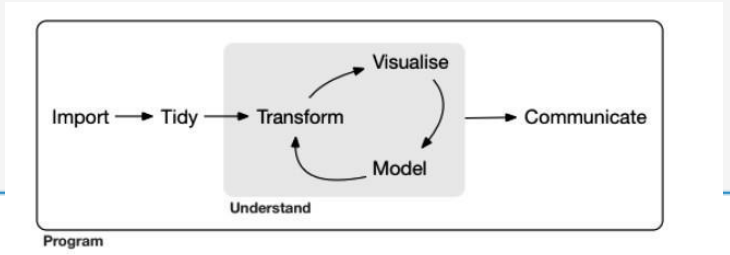
Possiamo anche definire altri operatori binari, così creiamo la soma al quadrato:

```
'%()%' =  function(x,  y) {(x +  y)^2}
2  %()%  3
```

3.1.8 PACCHETTI

Ultima cosa per i pacchetti:

```
#  installed  packages
(.packages(all.available=TRUE))

#  loaded packages
(.packages())

#  install a package
install.packages("igraph")

#  load  a package
library(igraph)
```



Per l'help:

```
?log
?'+'
#  Per cercare la  parola regression dei comandi dell'help:
??"regression"
```

Quando usciamo da R attenzione che nella cartella ci saranno due dati r.data e r.history questi sono l'ambiente e la storia dei comandi. Possiamo mettere se salvare in automatico o meno l'ambiente. L'ambiente è una tabella che associa le variabili ai nomi, la storia invece è la storia dei comandi che abbiamo digitato. Questo è interessante, ma pericoloso perché magari ci troviamo in un ambiente che avevamo creato mesi prima e se salviamo una variabile che pesa tanto diventa una rottura, conviene cancellarla.

APPLICATION

http://adv-r.had.co.nz/Subsetting.html#applications

3.1.9 RANDOM

Pseudo casuali e non casuali perché si avvicinano molto ad essere causali ma essendo l'aritmetica di un computer finita non possono essere troppo teorici.

Nota: se mettiamo una espressione dentro la parentesi allora oltre all'assegnazione anche viene stampato l'elemento creato

3.2.10 TIBBLES

Studiare dal libro pag 119

Ora introdurremo tibbles che migliora leggermente la struttura DB e vedremo ora in che modo. Esso è presente in tibble. Per tradurre un DB in un tibble basta fare

```
as_tibble(iris)
```

Oppure per creare un tibie la sintassi è simile a quella del DFrame

```
tibble( x=1:5, y=1,
        z=x^2 +  y
)
```

Una cosa che possiamo fare con i tibble è dare dei nomi che non sono identificatori validi, cosa che non si può fare con i DFrame. Dato un tibble possiamo trasformalo in un DFrame con as.dataframe?

Ci sono molte differenze tra tibble e DFrame. In tibble non posso creare una tabella con due colonne con lo stesso nome. Il DFrame lo lascia fare ma cambiando il nome della seconda colonna NomeColonna.1 .

Tibble non fa mai partial matching, cioè posso scrivere l'inizio del nome della colonna e vado a cercare la colonna che inizia in quel modo.

Iniziamo ora il ciclo della Data Science: importazione dei dati, tidy, fase ciclica: trasformazioni visualizzazione model, comunicazione.

Da adesso in poi vedremo una carrellata di tutti questi passaggi. La prima fase è

3.2 IMPORTAZIONE DEI DATI

Abbiamo i dati in un qualche formato e li vogliamo legger e scaricare in memoria. Gestiremo questa parte con il pacche readr. La funzione più importante è la read_csv() che legge appunto i file di tipo csv. RStudio prende le prima 1000 righe del file e specifica il tipo come succede nel libro. Posso aggiungere dei parametri a read_csv che alcune volte mi servono. Il parametro skip = n salta le prime n righe, poiché magari c'è un commento.

Se mettiamo comment="#" diciamo di saltare tutti i commetti. Se c'è un file senza intestazione meno col_names=FALSE legge il file senza intestazione. Oppure posso introdurre io i nomi. Una cosa molto importante l'interpretazione dei valori NA. Se abbiamo un file csv in cui il carattere na è definito tramite il ".", allora si dovrà specificare scrivendo na="." , se no immette il punto.

chal<-read_csv("challenge.csv",)

Questo csv ha dei problemi perché i primi 1000 tuple sono di tipi diversi da quelle successiva, cioè la prima colonna prima è integer e poi double e la seconda prima ha solo NA e successivamente solo date. Il View mostra infatti una tabella incompleta, questo esercizio ci porta a pensare che se sappiamo i tipi di colonna allora subito le definiamo. Allora possiamo fare così:

```
# Definiamo così  i tipi
chal<-read_csv("challenge.csv",  col_types=cols(x=col_double(), y=col_date()))
# Giochiamo sporco  e facciamo controllare  una riga in più
> chal<-read_csv("challenge.csv",  guess_max=1001)
```

Possiamo anche scrivere un csv, c'è anche un modo per salvare il tipo delle variabili.

Non sempre i dati da caricare sono semplici come sembra, per far fronte a ciò vediamo un esempio. Nell'esempio c'è un prologo e successivamente la tabella vera e propria.

APPLICATION

The goal of this make is to read data on wins and losses for all World Series games.

1. Read R documentation for function [redacted]. In particular pay attention to attributes what, skip, and nlines
2. Use s[redacted]o read data on wins and losses for all World Series games
3. The function s[redacted]reads from left to right, but the dataset is organized by columns and so the years appear in a strange order. Solve this problem (Hint: use [redacted] function).
4. Make a data frame with year and pattern

```
x<-scan(file=url("http://lib.stat.cmu.edu/datasets/wseries"), what=list(year=integer(0), pattern=character(0)),  skip=35, nlines=23)
x1<-order(x$year)
x2<-data.frame(year=x$year[x1], pattern=x$pattern[x1],  stringsAsFactors =  F) View(x2)
```

Capiterà spesso di beccare DB fatti così quindi conviene sapere bene come usare R base e in particolare scan.

3.3 TIDY

Vuol dire ordinato che è la parola che da il nome a tidyverse, che contiene diversi pacchetti. L'idea è quella di creare una funzione che ordini dei dati disordinati. Una variabile è una quantità o una qualità che possiamo misurare. Un valore è lo stato di una variabile. Una osservazione è un insieme di misure fatte sotto simili condizioni. Ogni osservazione contiene un numero alto di valori associate a differenti variabili. Sulle colonne mettiamo le variabili sulle righe mettiamo le osservazioni, le celle sono i valori. Noi vogliamo i dati in questo modo perché un formato uniforme consente:

• **consistenza**: facilità di apprendimento degli strumenti (ggplot, dplyr, ….) che fanno l'ipotesi che i dati abbiano questa forma e di conseguenza avendo una forma consistente è più facile creare ed usare questa strumenti.
• R è vettoriale ed avere le variabili come colonna ci permette di sfruttare a pieno questa caratteristica di R.

Alle volte ci dimentichiamo che l'analisi è più importante dell'importo dei dati. Quindi bisogna importare ed organizzar i dati per l'analisi e non viceversa. Spesso si cerca di facilitare l'entrata del dato e non per la sua analisi. Bisogna mettere i dati nella forma più comoda per l'analisi. Si spenda un po' più tempo per progettare lo schema dei dati ed inserirli secondo quello schema e senza dover perdere tempo poi per l'analisi. Ci sono comunque dei motivi per non usare il modello tidy. Ci sono rappresentazioni differenti già veloci. Un tipico adempio è la matrice di adiacenza del grafo, che una matrice quadrata dove la cela $i\ j - e\ s\ i\ m\ a$ vale 0 o 1 se c'è o meno l'arco tra i e j.
Supponiamo di voler organizzare: paesi, un anno, popolazione e una variabile casi (numero di casi come omicidi, ecc.). Ci sono diversi modi per creare una tabella, che dal punto di vista della teoria dell'informazione sono equivalenti, ma che dal punto di vista della teoria della normalizzazione si ha dei problemi con alcune tabelle. Ci sono due operazioni simmetriche sono gathering e spreading che ci servono per passare da una tabella disordinata ad ordinata. La prima operazione serve quando delle variabili sono di fatto dei valori come nella tabella 4 nel libro secondo questo esempio in cui gli anni sono dati e non colonne, in più si sta mettendo sulla stessa riga due osservazioni (osservazioni 1999 e 2000), questo non va bene per la definizione di righe nel tidy, quindi bisogna trasformarle.

```
table4a %>%
  gather(`1999`,  `2000`,  key =  "year",  value =  "cases")
```

Si mette prima il nome delle colonne da spostare in valori e poi il nome della nuova colonna e che valori diventano quelli che appartenevano alla variabile year.

Lo **spreading** invece va a risolvere il problema della tabella due in cui nella colonna key conteneva delle variabili e non dei valori, e quello che vogliamo fare è spostare questi valori nella intestazione di due colonne. Non si osserva la regola per la quale una osservazione deve essere rappresentata su una riga.

```
spread(table2, key = type, value = count)
```

C'è un'altra tabella (la 3) in cui in una tabella si mischiano due informazioni. Per separare le informazioni si usa il comando separate, che si usa quando i valori non sono atomici.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
```

Il contrario è unire le colonne, cioè quando abbiamo un informazione parziale.

```
table5 %>%
  unite(new, century, year, sep = "")
```

Queste presentate sono le 4 primitive per questa filosofia di ordine.
Ora vedremo una applicazione di ciò che abbiamo visto: http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/make/tidy/who.html

Le prime tre colonne del DB who rappresentano la stessa cosa, quindi c'è una ridondanza. Per cui potremmo tenere solo la prima. Poi ci sono delle colonne che iniziano con new o old, poi c'è un separatore _ e poi ci sono due o tre lettere che ci dicono il tipo di diagnosi, poi un separatore _ e poi una lettera e i numeri che sono il sesso e il range di età. Quindi abbiamo 5 informazioni nella stessa colonna. Vogliamo trasformare il DSet per entrare nelle regole del tidy. Come prima cosa facciamo un gather:

```
who1 <- who %>%
  gather('new_sp_m014':'newrel_f65', key = "key", value = "cases", na.rm = T)
who1
who2 <- who1 %>%
  mutate(key = str_replace(key, "newrel", "new_rel"))
who2
who3 <- who2 %>%
  separate(
key, c("new", "type", "sexage"), sep = "_")
who3
who4 <- who3 %>%
  separate(sexage, c("sex", "age"), sep = 1)
who4
who4 %>% count(new)
who5 <- who4 %>%
  select(country, year:cases, -new)
who5

who6 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", na.rm = T) %>%
  mutate(key = str_replace(key, "newrel", "new_rel")) %>% separate(key, c("new",
  "type", "sexage"), sep = "_") %>% separate(sexage, c("sex", "age"), sep = 1)
  %>% select(country, year:cases, -new)
who6
```

Per dividere età inferiore e età superiore:

```
who7 <- who5 %>%
  separate(age, c('InfAge', 'SupAge'), sep=-2) View(who7)
```

3.3 TRANSFORM

Fatta l'importazione dei dati possiamo partire con la parte fondante di tidyverse. Ora siamo nel blocco centrale delle fasi della datascience. Vediamo ora l'operatore Pipe e le sue alternative. Il pipe ci serve per accorciare la distanza tra quello che diremmo e le frasi che utilizzeremo per fare il processo di trasformazione e il linguaggio che faremo al pc. Avere un linguaggio di programmazione che ci permette di scrivere questioni vicino al linguaggio naturale è molto comodo. Ci sono tuttavia dei casi in cui il pipe è meglio non utilizzarlo. Se abbiamo comandi molto lunghi è meglio spezzarli per facilitare il debug. Il pipe è un operatore lineare, cioè ci permette di formare delle sequenze, quindi si sposa bene con i operatori unari (select, order, ….) e non con quelli binari come il join.

Il pacchetto che contiene il pipe è magrittr, oltre a %>% c'è anche %T>% che ritorna il lato sinistro e non il destro.

```
rnorm(100)  %>% matrix(ncol  =
  2)  %>% plot() %>%
    str()
#>    NULL

rnorm(100)  %>% matrix(ncol  =
  2)  %T>% plot() %>%
    str()
#>    num  [1:50,  1:2]  -0.387 -0.785 -1.057 -0.796 -1.756 …
```

Cioè con la T non passo il plot, ma il matrix, quindi passa la parte a sinistra e non a destra. Poi c'è ancora la possibilità del dollaro %$%.

Entriamo nel vivo della trasformazioni con il pacchetto più famoso: dplyr. Che è una grammatica per la trasformazione dei dati. Esso contiene dei verbi, cioè delle funzioni, che servono per manipolare i dati. Questi comandi non fanno altro che simulare le operazioni che abbiamo nell'algebra relazionale e nel SQL. Sono chiamati verbi perché viene usato un nome che sta nella grammatica dei linguaggi naturali, infatti si parla di grammatica dei linguaggi naturali. Questi verbi lavorano in modo molto simile. Anzitutto lavorano su DSet che sono normalizzati secondo tidy. Hanno come argomento un DFrame e come output hanno un DFrame. Lavoreremo spesso con il DFrame die voli.

FILTER

Corrisponde alla clausola where del linguaggio SQL, essa è una operazione di selezione nell'algebra relazionale. Il primo argomento è il DFrame e il secondo sono i cosiddetti filtri, cioè formule logiche. Mettere & o la virgola , è uguale.

Nota: nella condizione booleana mai fare confronti su numeri reali (es. sqrt(2)^2==2 da FALSE) per problemi di approssimazioni, meglio usare near()

ARRANGE

Serve per ordinare le osservazioni.

SELECT

MUTATE

Consente di costruire alcune variabili. Queste si possono costruire in funzione delle variabili preesistenti. Bisogna ricordarsi che le fusioni di mutate devono essere vettorizzate, cioè che hanno come input un vettore e come output un vettore.

TRASMUTE

SUMMARIZE

Spesso usato con group by che in realtà può essere usato in modo indipednente. Esso prende un DFrame e calcola una statistica aggregata sui dati e da come risultato quella statistica. Se per esempio voglio calcolare la media dei ritardi dei voli

```
summarise(flights,  delay =  mean(dep_delay, na.rm =  TRUE))
```

Si noti che si tolgono i na, se no non funzionerebbe. Il risultato è un singolo numero che rappresenta la media.

```
#> # A tibble: 1 × 1
#>   delay
#>   <dbl>
#> 1  12.6
```

In media abbiamo 12 minuti di ritardo. Questo verbo è poco utile da solo, ma molto utile se usato con group by.

GROUP BY

Implementa elementi a seconda di alcuni gruppi. Esso raggruppa le tuple con uno o più attributi. Cioè ogni gruppo ha lo stesso valore per quel determinato attributo. Quello che viene dopo su un group by va a lavorare su i singoli gruppi creati. L'operazione del group by è prendere la tabella, dividerla in $G_1, \ldots G_n$ gruppi.

```
by_day <-  group_by(flights,  year,  month, day)
summarise(by_day, delay  =  mean(dep_delay, na.rm =  TRUE))
```

```
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day delay
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.55
#> 2  2013     1     2 13.86
#> 3  2013     1     3 10.99
#> 4  2013     1     4 8.95
#> 5  2013     1     5 5.73
#> 6  2013     1     6 7.15
#> # ... with 359 more rows
```

Per togliere la partizione sta fare:

```
by_day<-ungroup(by_day)
```

Esistono diverse funzioni per fare le mie statistiche aggregate oltre a mean, come sum(), n_distinct(), n(). n() è talmente frequente che si usa più spesso count(), che direttamente conta secondo i gruppi. Ad Esempio:

```
count(flights, dest)
```

Che conta a seconda della destinazione il numero di voli. Si può anche assegnare un mese come:

```
count(fligts,  tailnum, wt  =  distance)
```

Per cui si calcolano per aeroplano il numero di miglia percorso perché moltiplico ogni volta per la distanza.

```
flights %>%
filter(!is.na(dep_delay),  !is.na(arr_delay) %>%
group_by(year,  month, day) %>%
```

summarize(

```
n=n(),
count(sum(dep_delay >  60), prop =
mean(dep_delay >  60)
)
```

Con questo comando troviamo il numero e la proporzione dei ritardi maggiori di un'ora. Si ricorda che prima bisogna filtrare togliendo quelli che hanno valore nullo se no non funziona.

```
flights %>%
group_by(dest %>%
summarise(carriers =  n_distinct(carrier)) %>%
arrange(desc(carriers))
```

Così troviamo le destinazioni con più compagnie aree che ci arrivano. È facile usare la clausola filter dopo il summarize per fare l'having di sql. Per trovare le destinazioni più popolari.

```
group_by(flights, dest)%>%
summarize(count=n()) %>%
filter(count>365)
```

Un'altro discorso è applicare un filter o un mutate direttamente dopo il group_by. Cosa poco comune a utilizzata. Troviamo i dieci voli più in ritardo per ogni giorno, quindi per ogni gruppo!

```
group_by(flights,  year,  month, day) %>%
filter(min_rank(desc(arr_delay)) <=  10)
```

Come si può vedere dal DFrame risultante si hanno per ogni giorno i primi 10 voli più in ritardo. In caso di pari merito sono più di dieci.

INTERSEZIONE UNIONE E DIFFERENZA

Inizia con le operazioni binarie. Si ricorda che il risultato non sarà mai duplicati perché restituisce un insieme

JOIN

Anche qua c'è il join, e ci sono tutte le forme viste in SQL. Innanzitutto c'è l'inner join, cioè quello interno, per il quale si selezionano solo le tuple che soddisfano la condizione di join e sono presenti in entrambi le tabelle. La funzione di chiama inner_join(). Ci sono anche gli altri join: left, right, full.

Un'altra distinzione è il join naturale o meno. Quello naturale è su due tabelle combinando attributi dello stesso nome. Se vogliamo fare il confronto solo su alcuni attributi basta mettere by="tailnum" ad esempio. Se vogliamo ad esempio far confronto tra flights e airport scriviamo by=c("dest"="faa") perché hanno nome diverso ma raccontano la stessa cosa. Questo by va scritto nel comando del join. Esiste un tipo di filtraggio che è il semi_join che prende le tuple presenti nella tabella di sinistra e di destra. l'anti_join fa il contrario di semi_join perché prende le righe che non sono presenti nella tabella a destra.

Application

We already wrote the following queries in SQL. Now try in R (both base R and dplyr).
For base R, use the following functions:

- subset to filter rows and select columns;
- order to arrange rows;
- transform to add variables;
- aggregate to group rows;
- merge to join data frames.

Run the queries with respect to the dataset nycflights13. Before querying, add an surrogate key called id to

the data frame flights.

1. flights on Christmas

2. flights that have a delay (either on departure or on arrival)
3. flights that were not cancelled (that is, those with valid departure and arrival times)
4. flights that have a departure delay sorted by delay
5. flights that catched up during the flight sorted by catch up time
6. the number of flights per day
7. the busy days (with more than 1000 flights)
8. the number of flights per destination
9. the popular destinations (with more than 365 flights) sorted by number of flights in descending order
10. the mean departure delay per day sorted in decreasing order of all flights on busy days of July
11. flights that flew with a plane manufactured by BOEING
12. flights that flew to a destination with an altitude greater than 6000 feet sorted by altitude
13. flights that took off with a plane with 4 engines and a visibility lower than 3 miles
14. flights with destination and origin airports with an altitude difference of more than 6000 feet

```
# Aggiungamo l'id
flights <- mutate(flights, id = row_number()) View(flights)



flights %>%
    filter(day==25, month==12)
subset(flights, month==12 & day==25)

flights%>%
    filter(!is.na(dep_delay) | !is.na(arr_delay))


flights %>%
    filter(!is.na(arr_time), !is.na(dep_time))


flights %>% filter(dep_delay>0) %>%
    arrange(desc(dep_delay))
df=subset(flights, dep_delay >0)
df[order(desc(df$dep_delay)), ]


flights %>%
    mutate(catchup = dep_delay-arr_delay) %>% filter(catchup
    > 0) %>% arrange(desc(catchup))
df = transform(flights, catchup = dep_delay - arr_delay)
df = subset(df, subset = (catchup > 0), select = c (id, dep_delay, arr_delay, catchup))
df[order(desc(df$catchup)),]


flights %>%
    group_by(day, month, year)%>%
    count()
?aggregate
df = aggregate(id~day + month, data=flights, FUN = length)
colnames(df)[3] = "count"
df = subset(df, count > 1000)
head(df, 10)


flights %>%
    group_by(day, month, year)%>%
    count() %>%
    filter(n > 1000)
```

```
flights %>% group_by(dest)
    %>% count()

flights %>% group_by(dest)
    %>% count() %>% filter(n
    > 365)%>%
    arrange(desc(n))

flights %>% filter(!is.na(dep_delay)) %>%
    group_by(day, month, year) %>%
    filter(month == 7) %>%
    summarize(avg = mean(dep_delay)) %>%
    arrange(desc(avg))

inner_join(flights, planes, by = "tailnum")%>%
    filter(manufacturer == "BOEING")

df <- merge(flights, planes, by = "tailnum")
subset(df, manufacturer == "BOEING")

inner_join(flights, airports, by =c("dest"="faa"))%>%
                        filter(alt > 6000)%>%
                        arrange(alt)

inner_join(weather, flights, by=c("day", "month", "hour", "origin")) %>%
    filter(visib < 3)%>% inner_join(planes, by =
    "tailnum")%>% filter(engines == 4)


inner_join(flights, airports, by=c("origin"="faa")) %>% inner_join(airports, by =
    c("dest"="faa")) %>% mutate(altdelta=alt.y-alt.x) %>%
    filter(altdelta > 6000)
```

Abbiamo quindi visto la parte di trasformazione. Quindi con un breve confronto tra DB e Data Science:

Un data scientist deve conoscere il modello concettuale, relazionale e SQL. Il primo perché bisogna creare un buon schema dei dati prima di analizzarli, e SQL perché spesso i dati non sono disponibili in memoria centrale. Nel DB esistono le forme normali, che hanno una buona proprietà, nel DS esiste la normalizzazione tidy.

Per quanto riguarda i vincoli di integrità nei DSet sono un po' meno importanti anche se sarebbe meglio seguire le regole dei DB, almeno quello di chiave. Se non c'è una chiave naturale è possibile aggiungerla tramite mutate. È possibile verificare se il vincolo di chiave esterna è rispettato con un anti_join.

```
flights %>%
    anti_join(airports, by = c("dest" = "faa")) %>%
    count(dest, sort = TRUE)
```

E poi si sana questo vincolo con alcuni metodi che per esempio:

```
id_set = anti_join(flights, airports, by = c("dest" = "faa"))$id flights = mutate(flights, dest =
ifelse(id %in% id_set, NA, dest))
```

Il limite di dplyr è che non implementa le modifiche come insert, update e delete. Ciononostante si può simulare la situazione così:

```
library(nycflights13)
library(dplyr)

# a table of cities
city = distinct(flights, dest)
city = rename(city, code = dest)

# another table of cities
new = distinct(flights, origin)
new = rename(new, code = origin)

# the two tables are not disjoint intersect(city,
new)

# insert as union
city = union(city, new)

# delete as set difference city =
setdiff(city, new)

# update as mutate
city = mutate(city, id = row_number()) %>% select(id, code)
# update a single row
mutate(city, code = ifelse(code == "ACK", "AAA", code))
# update manu rows
mutate(city, code = ifelse(id %in% c(1, 2, 3), "AAA", code))
```

C'è una corrispondenza tra il join di SQL e di dplyr. In DS il join di default è il left_join a differenza di SQL che quello di default è l'inner_join, poiché in DS non si vuole perdere informazioni. IN più semi e anti join possono essere utili.

Infine le tabelle annidate non esistono in SQL. In generale vedremo più avanti le colonne a lista, cioè colonne che contengono dei DFrame.

3.4 Visualizzazione

Per eliminare il par dopo che si hanno fatto delle modifiche:

```
par_old <- par
par(color=red)
par_old
```

In questo modo dopo aver fatto grafici di color rosso, possiamo ristabilire i parametri iniziali richiamando par_old.

Sia plot che ggplot funzionano con il **modello della tavolozza**, cioè aggiungiamo strati che non si possono togliere, si aggiungono così degli strati, però non si può tornare indietro. ggplot2 implementa una grammatica di visualizzazione dei dati. Anche qua come tidy abbiamo le medesime definizione di variabile, osservazione e valore. Esistono tre componenti principali di un grafico in ggplot:

- **dataframe**: ogni volta che si costruisce un grafico bisogna fornire il DFrame. Questa è una differenza sostanziale rispetto a plot che non richiede il DFrame.
- Associazione delle **estetiche**: associa le variabili e le estetiche del nostro grafico. Una estetica. Ia proprietà visuale di un oggetto del nostro grafico (forma dei punti, colore, dimensione delle linee, posizionamento dei punti, …).
- Oggetti **geometri**: rappresenta la semantica del nostro grafico (barplot, scatterplot, …)

Ovviamente le estetiche come gli oggetti sono fortemente legati, perché se per esempio abbiamo oggetti punti allora avremo estetiche per i punti e così via.

In generale:

```
ggplot(data =  <DATA FRAME>)+
              <GEOMETRIC OBJECT>  (mapping =  aes(<MAPPINGS>))
```

Vediamo ora alcune estetiche, si segue molto la linea guida del libro.

Noi vedremo le forme geometriche del grafico analizzando la variazione e la covariazione. Ci si ricordi del comando estetico binwith che ci da l'ampiezza della classe per l'istogramma. Se metto nell'estetica y=..density.., si pone la frequenza relativa.

Se abbiamo il confronto tra due variabili quantitative per analizzare la covarianza è lo scaterpot, essendo che questo non mostra benissimo la frequenza spesso si usa il comando di geometria geom_hex.

Si possono usare anche più geometrie alla volta.

Come nei linguaggi di ggplot c'è il concetto di variabile locale e globale. Se io metto le estetiche in ggplot() essa è globale e vale per tutte le geometriche successive, se si mette l'estetica in una geometrica vale solo solo in quella, quindi è locale. Anche il dataset è globale o locale. Si basti provare questi comandi:

```
ggplot(data =  mpg, mapping  =  aes(x =  displ, y  =hwy, col  =  drv))+
                 geom_point()+
                 geom_smooth()


ggplot(data =  mpg, mapping  =  aes(x =  displ, y  =hwy))+
  geom_point(aes(col =  drv))+
  geom_smooth()
```

Quando dobbiamo esporre il nostro lavoro dobbiamo lavorare un po' più fino dal punto di vista estetico, a partire dal nome delle assi, della leggenda, ecc..

Vediamo ora la parte estetica di R con la data challenge phyllotaxi. Il markdown si trova nella cartella degli appunti, in una sua cartella specifica chiamata "Phyllotaxi".

3.5 PROGRAM

Nella creazione di una funzione si ricorda che l'ultimo comando è quello preso come risultato dell'espressione.
L'ordine dei parametri per invocare una funzione è importante.

3.6 ITERAZIONE

Si usa raramente perché R è un linguaggio vettoriale. Il 90% delle volte non serve, anzi l'iterazione rallenta l'algoritmo pesantemente.

3.7 MODELLI

Noi vedremo solo la regressione lineare e vedremo una metodologia che può essere applicata a diversi modelli, e poi vedremo come lavorare con molti modelli (es. analizzare l'aspettativa di vita e l'istruzione in Italia o in tutti i paesi del mondo, in questo ultimo caso ci sono molti più modelli).
L'obiettivo di un **modello** è quello di rappresentare in maniera semplice un DSet ad una dimensione contenuta. Parleremo di **segnali** e **rumori**. Sappiamo che quello che cattura un modello è il **pattern** (segnali) e trascura il rumore, cioè la variazione causale che non ci interessa. I modelli tralasciano tutta una parte di definizione dei dati che denomineremo rumore. Ciò che è segnale e ciò che è rumore dipende dai dati e non dal modello, cambiando i dati queste qualità cambiano maniera suscettibile. Noi non vedremo un approccio

teorico, ma generalizzeremo ipotesi per modelli e non la conferma. Generalmente utilizzeremo i modelli per creare delle ipotesi e non per confermarle. E useremo dei DSet artificiali e non reali.

Nella seconda parte (Model Building) useremo DSet reali, e applicheremo ciò che abbiamo analizzato del model Basic. Nella terza parte impareremo a lavorare con una quantità di modelli superiore all'unità sfruttando i'iterazione funzionale che abbiamo analizzato nel capitolo precedente. Sfrutteremo due ingredienti: List colums nei DFrame (concetto visto precedentemente, strutture annidate) e l'iterazione funzionale. Questi due strumenti sono molto strumenti che ci permettono di gestire in maniera elegante ed efficiente i casi in cui ci troviamo di fronte a centinaia di modelli.

Applicheremo un'approccio qualitativo e non quantitativo, non andremo quasi mai a misurare quantitativamente quanto il nostro modello è buono, ma ci fideremo di un giudizio qualitativo o del nostro scetticismo. In fine useremo dei modelli per fare statistica esploratoria.

MODEL BASICS

$$DATA = Pattern(Signal) + Residual(Noise)$$

MULTI MODELLO

Abbiamo visto la data challenge, si trova la lezione nel markdown della cartella gapminder - multimodelli.

4. Network Science

Non tutti i dati hanno una struttura tabellare, ma ci sono: le reti, i dati gerachici e il testo. Le reti possono essere rappresentati tramite un DFrame per i nodi e per gli archi, ma tipicamente viene utilizzata un'altra matrice dei dati che è la matrice di adiacenza per i grafi.

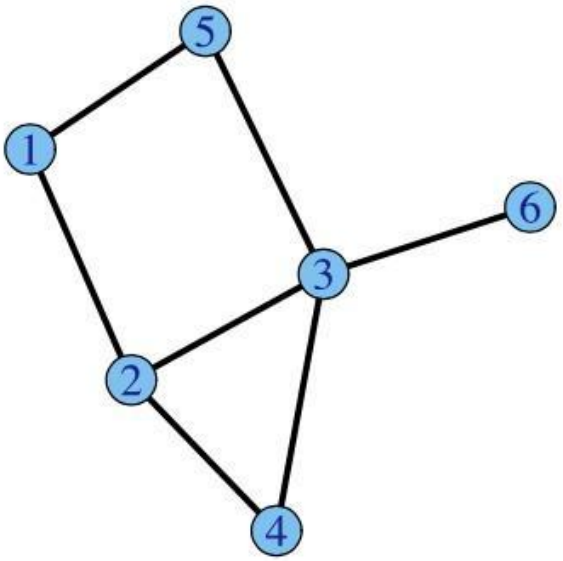
I dati gerarchici sono semi strutturati, cioè hanno una struttura ma non ben strutturata da far parte di una tabella (html, eccc.).

In fine il testo, se prendiamo un libro esso è completamente diverso da una tabella, anche se vedremo un approccio che si basa sulla struttura del DFrame.

Dovremo per cui analizzarli con un approccio a se stante rispetto il tidy.

NETWORK

Un grafo o una rete, sono sinonimi, è una collezione di nodi ed archi. Tipicamente un grafo in matematica è definita da una matrice matematica, chiamata matrice di adiacenza. Essa sarà una matrice e conterà nella $i\,j - e\,s\,i\,m\,a$ posizione 1 se c'è un legame tra l'elemento $i - e\,s\,i\,m\,o$ e $j - e\,s\,i\,m\,o$, zero se non c'è.



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Ci possono essere relazioni ad una direzione o a due direzioni. Relazione ……

GRAFI SEMPLICI O GRAFI MULTIPLI

I grafi multipli sono quelli in cui ci sono molti archi, ad es. le strade che uniscono due città (possono esserci più strade tra due città). Quelli semplici sono quelli visti fino ad ora. In forma tabellare i grafi multipli vengono rappresentati non limitando la cella ai due valori 1 e 0, ma si mette il numero di archi.

Ci sono i grafi pesati, su cui ogni arco viene definito assieme ad un peso. La matrice di adiacenza è definita con celle che hanno come numero il peso o 0. Il supporto dei pesi è R, quindi ci possono essere anche numeri negativi e di tutte le ampiezze.

CAMMINO E CICLO

Un cammino su un grafo è una sequenza di nodi congiunti da archi, seguendo ovviamente le direzioni. I cammini con nodi ripetuti sono chiamati non semplici, se no vengono vengono chiamati semplici. Un ciclo è un cammino che che ha come nodo iniziale quello finale. L'ordine è importante nelle sequenze, e quindi nei cammini e nei cicli.

Il cammino minimo su un grafo non pesato, è un cammino con minor numero di archi tra due nodi. Su un grafo pesato, il peso del cammino è la somma dei pesi degli archi, quindi si dirà minimo se è di peso minimo, in sostanza bisogna considerare i pesi. Non è detto che non esista un cammino unico o che esista.

Un grafo indiretto è connesso se esista una unica componente connessa, cioè se per tutte le coppie dei nodi esiste un cammino di minimo che li connette (semplicemente se non ci sono due grafi separati diciamo).

Un grafo diretto si dice fortemente connesso se per ogni coppia di nodi i e j esiste un cammino da i a j e un cammino da j a i (ad esempio un grafo diretto ciclico).

Si chiamano componenti connesse il massimo insieme dei nodi che sono connesse in un grafo. Un grafo è connesso se ha un'unica componente connessa.

I grafi bipartiti sono dei grafi per i quali si possono suddividere i nodi in due gruppi, tali che gli archi vanno solo da nodi di un tipo a quelli dell'altro. I grafi bipartiti sono degli esempi di grafi indiretti per i quali solitamente si usa un'altra matrice che è chiamata di incidenza: se i due gruppi hanno cardinali n e k, la matrice avrà ampiezza $n \times k$. In questo modo le righe saranno gli elementi di un gruppo e le colonne dell'altro, e si usa la medesima idea di porre 1 o 0 nelle celle se esiste o meno la relazione tra due elementi.

GRADO E GRADO PESATO

Il grado di un nodo nel caso di grafo indiretto è il numero di nodi vicini, cioè quelli che sono raggiungibili con un arco.

La nozione di grado si duplica nel caso di grafi diretti:

- il grado uscente: il numero di nodi che posso raggiungere da un nodo. Numero dei successori. Nella tabella si somma la riga.
- Il grado entrante: il numero di archi che mi portano al nodo. Numero dei predecessori. Nella tabella si somma la colonna.
- grado totale: la somma dei precedenti.

Se si ha un cappio quello sarà sia successore che predecessore.

Se abbiamo un grafo pesato si definisce il grado pesato, quindi si sommano i pesi e non i numeri di archi. Nel caso di grafi diretti avremo un grado uscente pesato e un grado entrante pesato.

Abbiamo visto fino ad ora dati che possono essere archiviati in forma tabellare, ma ci sono dati che "sfuggono" a questa griglia: testo, dati semi-strutturati e grafi/reti.

5 categorie per modellare i dati nel modo reale:

1. Le reti tecnologiche: internet
2. Le reti sociali
3. Reti di informazione: come il web
4. Reti di citazioni fra
5. Reti biologiche: interazione proteina-proteina, reti alimentari

Modellare i dati come insieme di punti collegati fra di loro, quello che conta è la relazione fra i nodi, il punto senza la relazione non è molto significativo.

Tramite il pacchetto igraph di R analizziamo le reti, tramite diverse misure misure di centralità, cammini minimi. Con ggraph invece visualizziamo le reti con un meccanismo simile a ggplot, quindi aggiungendo geometrie per archi e nodi.

Nelle reti pesate gli archi hanno associato un numero reale che ne quantifica la forza della relazione, se è negativo allora la relazione è inversa. Possiamo avere pesi nulli, ovvero assenza di relazione.

Una delle caratteristiche fondamentali delle reti è che sono *webs without spiders*, non c'è un'unità centrale che coordina la rete. Il tutto viene in modo organizzato autonomamente, gli agenti si auto-organizzano. La numerosità degli individui, più il comportamento degli agenti molto semplice può creare una proprietà visibile a livello globale. C'è un dunque un passaggio da micro a macro, interessante nelle reti complesse.

Misure di centralità

È una funzione che assegna un rating, un numero, ad ogni nodo. Con centralità si intende l'importanza di ogni nodo. Una volta assegnato un rating si può determinare un ranking. Google usa un algoritmo chiamato *pagerank* per ottenere i risultati in un certo ordine, per prime le pagine con centralità maggiore. Su internet potremmo cercare di capire quali sono le macchine più importanti per salvaguardarle da un attacco terroristico ad esempio. La nozione di centralità non è univoca. Ogni misura da l'importanza a un determinato aspetto della rete, è bene non fissarsi su una singola misura ma considerarne tante.

La prima misura di centralità è il **grado** di un nodo un nodo è importante se ha molte connessioni, è una misura semplice e abbastanza affidabile, ad esempio nelle reti di citazioni si usa per contare le citazioni. Nel caso di reti dirette abbiamo due misure: **grado uscente** e **grado entrante**. Dobbiamo quindi specificare il tipo di grado, anche se in realtà conta molto di più il nodo entrante, visto che è meno controllabile.

Un'altra misura di centralità è il **grado pesato**. Questa misura di centralità si usa nelle reti pesate, il grado pesato o **forza** è la somma dei pesi degli archi che raggiungono un determinato nodo.

Come esempio di studio useremo la rete terroristica che ha organizzato gli attacchi a Madrid nel 2004.

Closness

Reciproco della distanza media, dove con distanza media si intende la distanza geodetica.

Betweenes

Ci possono essere degli archi ad alta intermediazione, archi attraversati da molti cammini minimi. Rimuovendo questi archi spesso potremmo rimuovere il collegamento fra i due nodi.

C'è però una complicazione, lavoriamo su reti pesate. Il concetto di cammino minimo si estende anche su reti pesate. Il cammino è la somma dei pesi degli archi percorsi per raggiungere un nodo. Nella betweeness però si inverte il peso degli archi, così archi con peso basso rappresentano nodi vicini.

Cosa vogliamo fare con il peso e i cammini. Granovetter dice che se prendiamo una rete sociale si formano delle comunità molto coese, connesse da archi sporadici. Gli archi all'interno della comunità sono chiamati archi forti, *strong ties*, gli archi fra le comunità sono chiamati archi deboli, weak ties. Un legame forte è una relazione fra persone che condividono tante esperienze, e necessitano per rimanere attive di molta cura. Questi due legami hanno un'importanza molto differente. Si vede che i legami forti tendono a generare idee dominanti e stagnanti, viceversa i legami deboli tendono a favorire la diversità delle idee e un modo diverso di pensare. Secondo Granovetter sono più importanti gli archi deboli, danno la possibilità di accedere a mondi, modi di pensare diversi da quelli della propria comunità. Ma la teoria vale anche per la rete terroristica del nostro esempio?

Similarità

Vogliamo capire se due nodi sono simili fra di loro. Trovare nodi simili è un problema interessante, ad esempio è utile trovare pagine web simili, amazon cerca spesso clienti simili. Il problema della similarità è un problema binario, abbiamo due nodi e vogliamo capire quanto siano simili fra di loro. Diremo che due nodi sono simili se il pattern di connessione è simile. Dati due nodi S e T e un terzo nodo X, possiamo fare considerazioni di questo genere: se sono entrambi in relazione con X allora c'è più similarità, se S è in relazione con X e T no allora la similarità sarà maggiore.

Più simili sono le righe della matrice di adiacenza più simili sono i nodi. La similarità può essere misurata tramite il coefficiente di correlazione di Pearson.

Come potremmo visualizzare a similarità? Potremmo usare un dataframe di 3 colonne, due per avere from e to e una con la corrispondente similarità.

5. Dati semistrutturati - XML

Affrontiamo questo argomento perché ci stiamo muovendo sempre più verso dati poco strutturati. Abbiamo visto fino ad ora dati che possono esser rappresentati su una struttura "rettangolare". Con i network abbiamo visto sempre una struttura tabellare, dove però le colonne non erano le variabili e le righe non erano le osservazioni. Ci stiamo allora sgretolando sempre di più, come si suol dire "decomponendo" nell'ambito psicologico. Dobbiamo esser in grado alla fine di questo corso di analizzare si adati con una struttura che senza struttura, perché è facile analizzare dati strutturati, risulta più difficile farlo con dati meno strutturati.

Vedremo ora come analizzare una pagina web, per la quale non ci è mai venuto in mente di analizzarle in una tabella. Vedremo che ci sono molte varianti: assenza di informazione, informazione ripetuta, struttura gerarchica annidata. Ci sono due formalismi, noi vedremo XML, utilizzati per rappresentare tali dati.

XML è un acronimo e sta per *Extensible Markup Language* e quindi è:

- Un linguaggio formale. Cioè un documento XML è definito in base a delle regole formali, cioè una grammatica che dice esattamente come struttura un documento
- Markup significa annotazione e cioè i dati o il testo all'interno del documento sono annotati con dei tag, che annotato, cioè danno semantica a quel dato
- Extensible significa che è estensibile, cioè che l'insieme dei tag non è fissato a priori, ma definito dal progettista. A differenza di HTML dove i tag sono prefissati e non possono essere inventati.

Un documento XML è un documento di testo con delle annotazioni che servono per dare una semantica, cioè un significato all'informazione. Non è un linguaggio di presentazione come HTML, in particolare il significato dei tag non è di fare una formattazione, ma di dare un significato. Non è un linguaggio come java, non è un protocollo e neanche un DBase. È semplicemente un file di testo per rappresentare dati semistrutturati.

Vediamo ora la struttura di un documento XML. Parleremo innanzitutto del testo. Un testo sarà delimitato con dei tag o etichette come quelle che si usa per HTML. Per dire che Andrea Pesce è una persona scrivo:

```
<person>
     Andrea Pesce
</person>
```

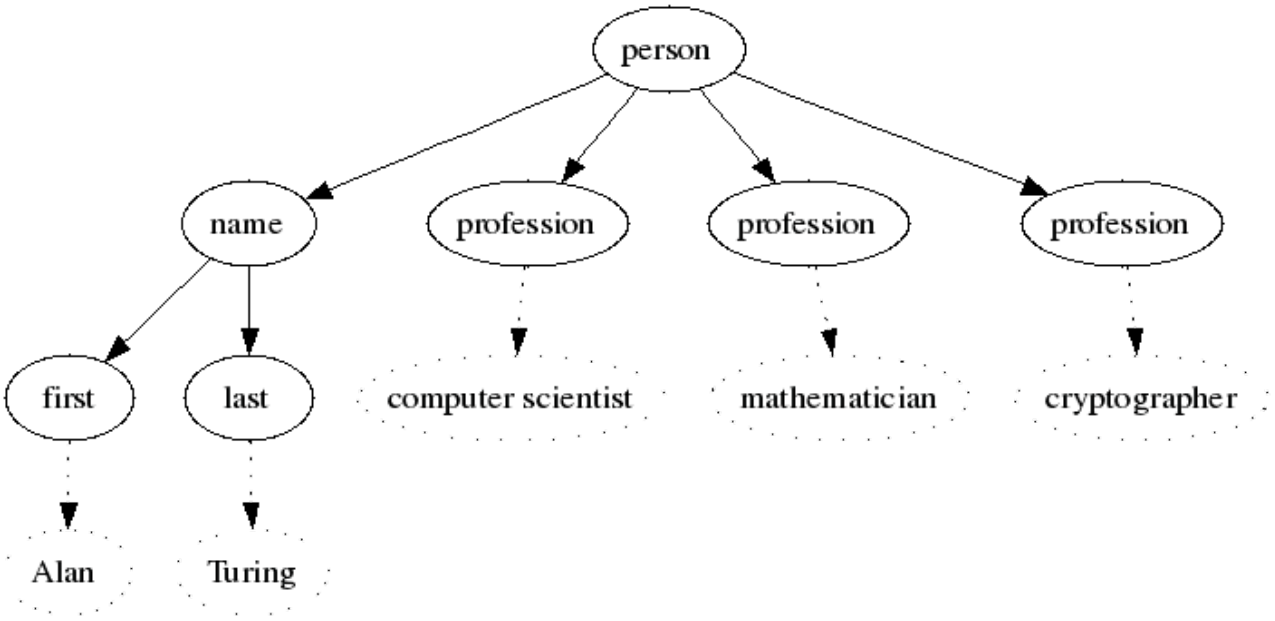
Si nota quindi la differenza tra il tag di apertura e il tag di chiusura. Questo è un **elemento**, Andrea Pesce è il suo **contenuto** e il **nome** dell'elemento è person. In questo modo sto dando semantica, cioè do un significato alla stringa Andrea Pesce. I tag servono a descrivere il tipo del contenuto. Vediamo un elemento più complesso:

```
<person>
  <name>
     <first>Alan</first>
     <last>Turing</last>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

È superfluo spiegare l'elemento perché i tag parlano da se. Si note che qui abbiamo degli elementi che contengono del testo ed altri elementi che sono struturati, cioè contengono altri elementi. Quindi una cosa importante di XML, è che un modello gerarchico, nel senso che n elemento può contenere elementi e che possono contenere altri elementi e così via. Però ci sono due regole:

- Gli elementi non possono sovrapporsi, cioè se apro persona e poi apro name, non posso chiudere person prima di name. La relazione tra gli elementi può essere o di inclusione, come name e person, oppure disgiunti come name e professione. Ma non possono essere intersecanti.
- Ci deve essere un elemento contenitore, cioè un elemento radice che contiene tutti gli altri. Questo fa si che la struttura è una struttura ad albero (grafo diretto privo di cicli). Ad esempio:

IN questo albero abbiamo i così detti nodi figli e nodi padre. Posso ance mescolare testo ed elementi. Ad esempio:




```
<person>
   <first-name>Alan</first-name> <last-name>Turing</last-name> is mainly known
   as a <profession>computer scientist</profession>. However, he was also
   an accomplished <profession>mathematician</profession>
   and a <profession>cryptographer</profession>.
</person>
```

Posso anche scrivere anche degli elementi vuoti abbreviando in questo modo:

```
<address/>
```

Si noti che XML è case sensitive.

Ci sono due componenti in un documento XML, uno sono gli elementi e gli altri sono gli attributi, che sono proprietà che valgono per l'intero elemento e si scrivono con questa sintassi. Ipotizziamo di voler aggiungere due attributi all'elemento person: Born e died. Che contengono la data di nascita e di morte di Alan Turing:

```
<person born="23/06/1912" died="07/06/1954">
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   <profession>mathematician</profession>
   <profession>cryptographer</profession>
</person>
```

La struttura quindi è *name = "value"*, l'ordine degli attributi è irrilevante e gli attributi all'interno di un medesimo elemento devono avere nomi differenti. Ora vediamo lo stesso documento in cui si una in modo smodato gli attributi, cioè si rappresenta tutta la informazione mediante gli attributi:

```
<person born="23/06/1912" died="07/06/1954">
   <name first="Alan" last="Turing"/>
   <profession value="computer scientist"/>
   <profession value="mathematician"/>
   <profession value="cryptographer"/>
</person>
```

Gli attributi hanno sempre una struttura atomica, mentre gli elementi possono essere strutturati in modo arbitrario. Ci son altri elementi che non sono importanti per noi. Prima di Codd, che ha inventato il modello relazionale c'era il modello gerarchico, a cui è molto simile l'XML.

Dobbiamo definire uno **schema** per il documento XML, innanzitutto dobbiamo capire in che senso schema per un documento XML, nel senso che conosciamo nelle basi dati. Nella base dei dati avevamo detto che c'è una tabella, quindi c'è una istanza che ha pure uno schema, che era il nome delle colonne e dei vari tipi. Quindi avevamo detto che lo schema definisce la tabella e l'istanza definisce il contenuto, stessa definizione che intercorre tra classi ed oggetti nella programmazione orientata agli oggetti. Anche qui c'è questa differenza: i documenti XML sono le istanze, cioè i dati, gli schemi specificano questa semistruttura dei dati, nel senso che si tratta di dati semistrutturati i quali devono essere posti nel documento secondo certe regole, le quali sono definite in uno schema.

Ci sono due linguaggi per definire gli schemi XML:

1. DTD, Document Type Definition (che vedremo)
2. XMLSchema

DTD, che una definizione di tipo del dato, cioè di classe dei nostri oggetti, è interessante perché ci 8introduce le espressioni irregolari. Vedremo come è possibile scrivere degli schemi. DTD non è un'applicazione XML, quindi non usa il linguaggio XML, ma si usano queste clausole. Potremmo scrivere lo schema così:

```
<!ELEMENT person     (name, profession*)>
<!ATTLIST person      born CDATA #REQUIRED died CDATA #IMPLIED>
<!ELEMENT name       (first,last)>
<!ELEMENT first      (#PCDATA)>
<!ELEMENT last       (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

Si noti che:

- profession* si mette la stella per indicare che può avere più determinazioni
- (#PCDATA)> principalmente significa che sono elementi atomici e possono contenere solo testo, cioè sono foglie del nostro albero.

Il documento che abbiamo visto prima risulta essere valido secondo questo schema, cioè soddisfa questo schema, ma anche altri documenti potrebbero esser validi. Ad esempio se togliessimo le tre professioni esso soddisferebbe ancora questo schema, anche se ne aggiungessimo ancora una. Quindi uno schema è come se fosse un insieme infinito di documenti e diremo che un documento/istanza soddisfa uno schema se fa parte di quella collezione.

Andiamo nel dettaglio. La prima cosa che dobbiamo scrivere in un documento XML e che vogliamo convalidare secondo uno schema è la cosiddetta Document Type Declaration, che è semplicemente una riga che ci dice qual è la radice del nostro documento, nel nostro caso *person*, e dove si trova la DTD. Diremo che un documento XML è valido se soddisfa le regole di quell'espressione regolare, cioè fa parte dell'insieme delle istanze. Come regola generale, quando si carica un documento XML con un Browser non viene fatta la validazione. Viene fatto solo il controllo che il documento sia ben formato, cioè che rappresenti l'albero (tag chiusi, non intersezione, …). Si deve fare attenzione che appunto non va a validare il documento.

Iniziamo. Vedere ora le componenti principali della DTD. La DTD serve per vedere lo schema. Sapendo che XML è definito in base agli elementi e gli attributi. Quindi mi serviranno delle regole per definire gli elementi e la loro struttura e delle regole, giù semplici per definire il contenuto degli attributi, sono più semplici perché gli attributi non possono essere strutturati. La definizione/dichiaraizone **element** serve per dichiarare l'elemento

```
<!ELEMENT name content>
```

Vediamo degli esempi:

```
<!ELEMENT email (#PCDATA)>
```

Qua mi aspetto un unico elemento email che contiene del testo. Quindi non altri elementi.

```
<!ELEMENT contact (e-mail)>
```

Qua mi aspetto che il mio elemento contact contiene un'altro ed un solo elemento che si chiama email.

```
<!ELEMENT contact (e-mail | phone)>
```

Qua mi aspetto che il mio elemento contact può contenere o l'elemento e-mail o l'elemento phone, ma non entrambi

```
<!ELEMENT name (first,last)>
```

Qua ci aspettiamo che l'elemento name deve contenere l'emetto first seguito dall'elemento last.

```
<!ELEMENT image EMPTY>
```

Posso dire che ho un elemento vuoto.

```
<!ELEMENT image ANY>
```

Posso dire che ho un elmetto di forma qualsiasi. Infine la parte più interessante è l'iterazione, perché si può iterare la composizione con la scelta con questi tre operatori:

*
Zero or more instances are allowed
+
One or more instances are allowed
?
Zero or one instances are allowed

Vediamo alcuni esempi: supponiamo che l'elemento name è strutturato in questo modo:

```
<!ELEMENT name (first*,middle?,last+)>
```

I prossimo sono tutti dei documenti che soddisfano tale schema:

```
  <first>Samuel</first>
  <middle>Lee</middle>
  <last>Jackson</last>
</name>

<name>
  <first>Samuel</first>
  <first>Michael</first>
  <last>Jackson</last>
</name>

<name>
  <last>Jackson</last>
  <last>Keaton</last>
</name>
```

Se mettessimo due elementi middle o zero di last non sarebbero sai validi. La prossima definizione dice che il nome può avere qualsiasi numero d first, middle e last in qualsiasi ordine:

```
<!ELEMENT name (#PCDATA | first | middle | last)*>
```

Certe definizioni sono vietate, in particolare la prossima definizione che dice che il nome potrebbe essere o il nome di battesimo o un cognome o un nome di battesimo seguito da un cognome. Questo è sbagliato

perché il contenuto non è deterministico. Perché se i lvalidatore incontra un elemento firts a questo punto ha due possibilità: o l'elemento name è finito o si aspetta un elenco last. Vedremo meglio quando analizzeremo gli automi. Questo comportamento non è ammesso per motivi computazionali. Ci si ricordi che le DTD devono essere deterministiche.

```
<!ELEMENT name (first | last | (first,last))>
```

Quindi per adesso abbiamo visto come scrivere delle istanze con XML e degli schemi con DTD. Ora vediamo come definire gli attributi. Con la parola chiave ATTLIST dichiariamo il suo elemento, l'attristo e il suo tipo:

```
<!ATTLIST element attribute TYPE DEFAULT>
```

In realtà il tipo non il tipo nel senso che conosciamo, ma possiamo dire semplicemente che l'attributo contiene del testo (CDATA) o dei numeri (Enumeration) o un ID. Quest'ultimo caso è quello che più ci interessa perché quando un attributo è di tipo ID sostanzialmente corrisponde a una chiave nel senso delle base di dati. Quindi il valore di tale attributo deve essere univoco per tutti gli attributi all'interno di quel documento XML. In più abbiamo l'IDREF che corrisponde alla chiave esterna. In questo caso il vincolo deve soddisfare la richiesta per la qual ci deve essere un altro attributo di tipo ID che contiene tali valori. Poi abbiamo IDREFS che è sostanzialmente una lista di valori IDREF separati dallo spazio.

Oltre a definire il tipo per attributo bisogna definire anche un **default definition.** Ci sono quattro possibilità per questi default:

1. #REQUIRED per il quale il valore dell'attributo deve essere definito
2. #IMPLIED per il quale il valore dell'attributo è opzionale

Supponiamo di avere una relazione banking tra un customer e un count. Quindi abbiamo due entità e una relazione *Bank* e supponiamo che un cliente possa avere più conti correnti e che un account possa essere associato ad un unico cliente. Quindi non ci possono essere conti condivisi. In XML modelliamo la relazione in questo modo:

```
<?xml version="1.0"?>

<!DOCTYPE banking [
  <!ELEMENT banking    (customer*)>
  <!ELEMENT customer   (name, account*)>
  <!ATTLIST customer    id ID #REQUIRED>
  <!ELEMENT account    (bank,number)>
  <!ATTLIST account     id ID #REQUIRED>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT bank       (#PCDATA)>
  <!ELEMENT number     (#PCDATA)>
]>
```

Vediamo due istanze valide:

```
<banking>
   <customer id="C1">
   <name>Massimo Franceschet</name>
   <account id="A1">
      <bank>Fineco</bank>
      <number>34567</number>
   </account>
   <account id="A2">
      <bank>ABN AMRO</bank>
      <number>98672</number>
   </account>
   </customer>
```

```
   <customer id="C2">
   <name>Enrico Zimuel</name>
   <account id="A3">
      <bank>ING Bank</bank>
      <number>8909</number>
   </account>
   </customer>
</banking>
```

La prima istanza ha due conti, mentre il secondo ne ha solo uno. Come vediamo non si ha usato il meccanismo delle chiavi esterne perché non serve, in quanto si può usare il fatto che XML sia gerarchico per andare un elemento dentro l'altro, per cui il fatto che il conto corrente A2 sia del customer C1 lo vedo perché sta dentro l'elemento con id C1. Questo è comodo perché evita di inserire il vincolo di chiave esterna. Si nota che tutto funziona nell'esmepio perché la relazione è uno a molti. Supponiamo infatti di cambiare ora la semantica e che un account possa avere almeno un proprietario

```
<?xml version="1.0"?>

<!DOCTYPE banking [
   <!ELEMENT banking    (customer*)>
   <!ELEMENT customer   (name, account*)>
   <!ATTLIST customer    id ID #REQUIRED>
   <!ELEMENT account    (bank,number)>
   <!ATTLIST account     id ID #REQUIRED>
   <!ELEMENT name       (#PCDATA)>
   <!ELEMENT bank       (#PCDATA)>
   <!ELEMENT number     (#PCDATA)>
]>
```

e prendiamo lo stesso documento, quindi la DTD è la medesima, però il documento è un altro:

```
<banking>
   <customer id="C1">
      <name>Massimo Franceschet</name>
      <account id="A1">
         <bank>Fineco</bank>
         <number>34567</number>
      </account>
      <account id="A2">
          <bank>ABN AMRO</bank>
         <number>98672</number>
      </account>
   </customer>

   <customer id="C2">
      <name>Enrico Zimuel</name>
      <account id="A2">
          <bank>ABN AMRO</bank>
         <number>98672</number>
      </account>
   </customer>
</banking>
```

Chiaramente questo non va bene perché c'è una violazione del vincolo di chiave primaria in questo ho due attributi di tipo ID con un valore duplicato, quindi non posso fare questa cosa. Allora è fondamentale la soluzione di prima che funzionava perché ad ogni account era associato un unico cliente e quindi non potevo avere duplicati. A questo punto se vogliamo tenere la relazione molti a molti devo passare ad un approccio relazionale: devo creare un elemento customer e uno ID in questo modo:

```
<!DOCTYPE banking [
  <!ELEMENT banking    (customer | account)*>
  <!ELEMENT customer   (name, accounts?)>
  <!ATTLIST customer    id ID #REQUIRED>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT accounts   EMPTY>
  <!ATTLIST accounts   idrefs IDREFS #REQUIRED>

  <!ELEMENT account    (bank,number,owners)>
  <!ATTLIST account     id ID #REQUIRED>
  <!ELEMENT bank       (#PCDATA)>
  <!ELEMENT number     (#PCDATA)>
  <!ELEMENT owners     EMPTY>
  <!ATTLIST owners     idrefs IDREFS #REQUIRED>
]>
```

Vediamo quindi un esempio:

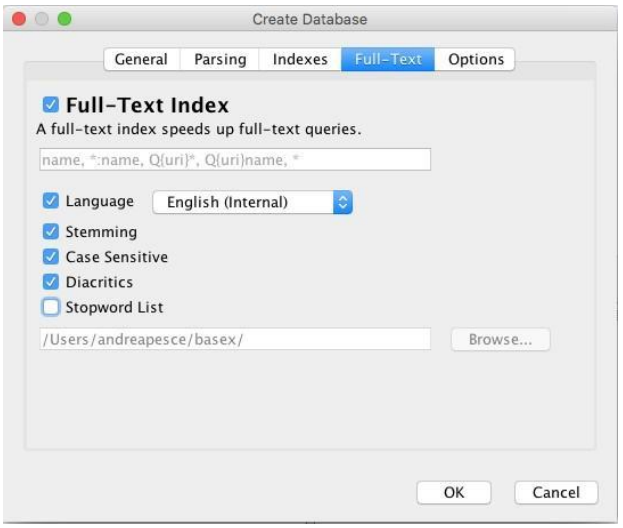
```
<banking>
   <customer id="C1">
      <name>Massimo Franceschet</name>
      <accounts idrefs="A1 A2"/>
   </customer>

   <customer id="C2">
      <name>Enrico Zimuel</name>
      <accounts idrefs="A2"/>
   </customer>

   <account id="A1">
      <bank>Fineco</bank>
      <number>34567</number>
      <owners idrefs="C1"/>
   </account>

   <account id="A2">
      <bank>ABN AMRO</bank>
      <number>98672</number>
      <owners idrefs="C1 C2"/>
   </account>
</banking>
```

Vediamo che ora le chiavi sono tutte univoche e quindi l'istanza è valide. In generale quando voglio modellare delle relazioni uno e uno o uno a molti posso sfruttare l'annidamento, mentre quando abbiamo relazioni di tipo molti a molti di deve sostanzialmente simulare il modello relazionale creando due mondi e connettendoli attraverso le chiavi esterne. Nel nostre esempio abbiamo connesso le entità da entrambi i versi, quindi per ogni cliente abbiamo una lista di account e per ogni account abbiamo fatto una lista di clienti. Ciò risulta un po ' ridondante e potrei fare solo una delle relazioni, ma ci viene comodo per applicare delle query. Se volgiamo ad esempio estrarre dato un cliente quali sono i suoi conti, chiaramente le chiavi questi conti sono immediatamente disponibili e viceversa.

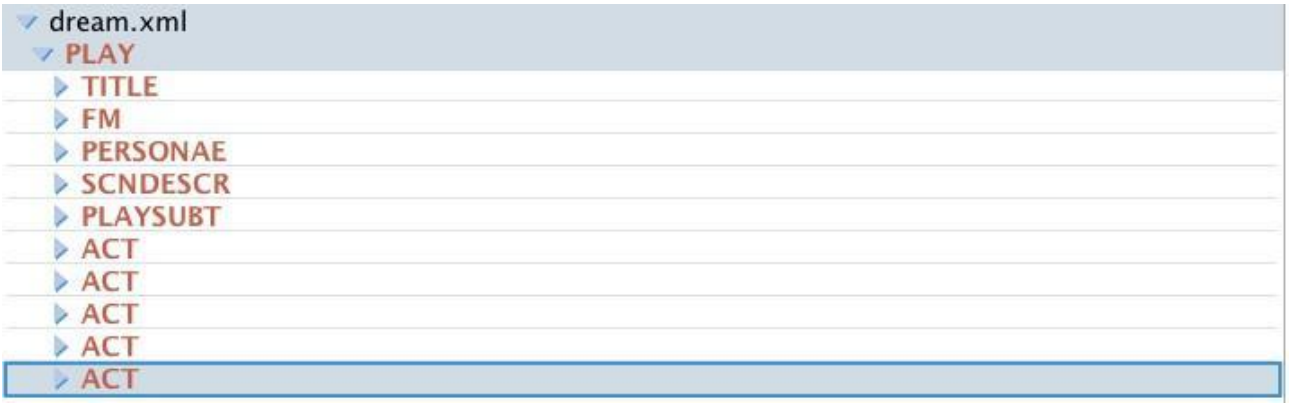
C'è un altro schema che è W3C che non vedremo perché è più complesso e formale, esso è utilizzato in particolare per i documenti XML più complessi. Noi andremo ad analizzare documenti semplici e ci limiteremo a tale livello.

Vedremo ora alcuni tipi di linguaggi di interrogazione di XML che ci permettono di estrarre informazione, un po' come avveniva con SQL. Questi linguaggi si chiamano XPath e XQuery che ci permettono di estrarre informazioni da un documento XML. Per vedere questi linguaggi useremo BaseX.

Aperto BaseX si vada su DataBAse e si carichi il file voluto, prima di mettere ok, si facciamo come nella figura seguente, in modo tale da creare degli indici per velocizzare le query:



Si ripresenterà l'interfaccia BaseX. In alto a destra c'è lo spazio dove immettere i comandi. Si possono cambiare le visualizzazioni del nostro schema. Nel nostro caso abbiamo messo la visualizzazione folder. Qua sotto abbiamo la figura che ci rappresenta la struttura del nostro XML.



Questa struttura è data grazie alla semantica di XML, che ci permette anche di interrogare tale struttura.

Vediamo ora la struttura del primo linguaggio che vedremo che si chiama XPath. XPath è un semplice linguaggio di recupero di elementi e attributi all'interno di un documento XLM, è una sorta di tecnologia jolly. Ogni nodo del nostro albero può essere o la radice dell'albero oppure è un nodo di tipo elemento o di tipo attributo o ci sono nodi più specifici. XPath è una sorta di espressione regolare su alberi, non su stringhe, e il risultato di una espressione regolare su XLM è un insieme di nodi. I nodi in un documento son ordinati così come appaiono in un documento, quindi se l'elemento A precede B nel documento, allora alo precederà anche nell'ordinamento. Si usa in questi casi l'ordinamento anticipato, che sarebbe l'ordinamento dei tag all'interno del documento. Per comprenderlo meglio conviene andare a vedere nelle slide in quanto spiegarlo a parole risulta complicato. È importante capire l'ordinamento perché le interrogazioni sfrutteranno tale ordinamento. Ipotizziamo di avere una radice con due figli i quali figli hanno altri due figli. Allora in ordine avremo la radice, poi il figlio di sinistra, poi i suoi due figli partendo sempre la sinistra, poi il secondo figlio della radice e in fine i suoi due figli partendo da sinistra.

La prima componente di una query XPath è una location step, cioè un passo che la una forma del tipo: axis::test[filter]. L'asse è una modalità per accedere ai nodi e il test son delle modalità per filtrare le parti che abbiamo trovato, il filtro è opzionale.

Vediamo la parte asse: l'asse più semplice sarà l'asse child, cioè che seleziona tutti i nidi figli. Quindi se siamo su un nodo e chiediamo mettiamo come asse child, avremo come risultato i suoi figli. Oppure l'asse parent assegnerà il nodo padre. Poi abbiamo l'asse descendant il cui risultato sono i nodi della discendenza, cioè i figlio, i figlio dei figli e così via, sostanzialmente in XLM tutti gli elementi che sono al'lintenro di tale elemento. L'asse ancestor invece prende gli antenati. E poi abbiamo right sibling (fratello sorella) per cui i nodi di destra che hanno lo stesso padre, mentre i fratelli di sinistra sono left sibling. Poi abbiamo l'asse follow che prende tutti i nodi che seguono quel nodo a parte i discendenti e precede prende tutti i nodi che precedono quel nodo a parte gli antenati.

Dopo aver visto l'asse vediamo il test che potrebbe essere: tipicamente. È il carattere star che sarebbe qualsiasi nodo di quel elemento ma potrebbe essere qualcosa di diverso, ad esempio un elemento specifico (ad esempio voglio tutti i figli che hanno quel nome) oppure altre cose se posse text() prendo tutti i figli di tipo testo, ecc.

Per esempio prendiamo il documento:

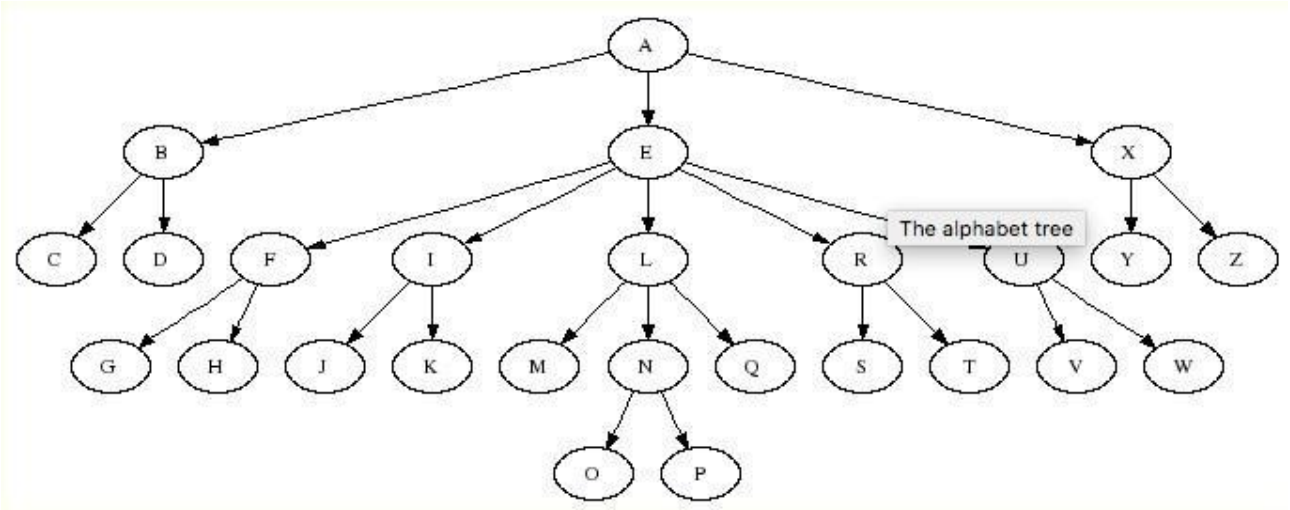
```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM "Turing.dtd">
<?xml-stylesheet type="text/css" href="Turing.css"?>
<!-- Alan Turing was the first computer scientist -->
<person born="23/06/1912" died="07/06/1954">
   <name>
      <first>Alan</first>
      <last>Turing</last>
   </name>
   <profession>computer scientist</profession>
   As a computer scientist, he is best-known for the Turing Test
   and the Turing Matchine.
   <profession>mathematician</profession>
   <profession>cryptographer</profession>
</person>
<!-- He committed suicide on June 7, 1954. -->
```

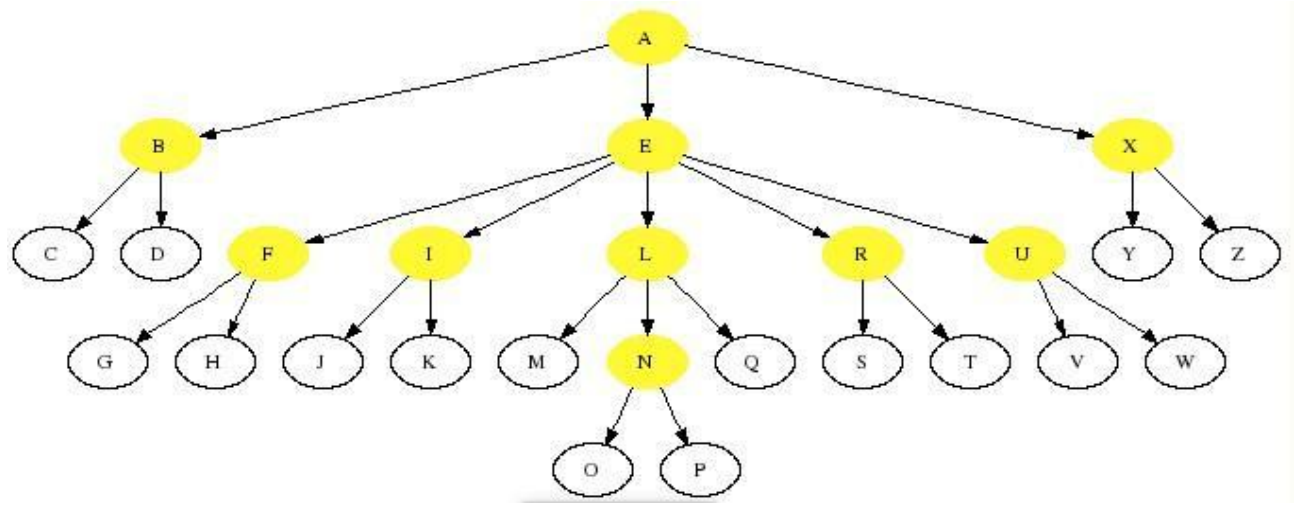
Supponiamo che il nodo con testo da cui valutiamo la nostra valutazione sia person e se ci chiediamo child::node(), che selezione tutti i figli di qualsiasi tipo allora verranno selezionati name, i 3 profession, ma anche il figlio di tipo testo "as computer….". Se invece scriviamo child::* selezionato solo i figli di tipo elemento: name e i tre profession. Se scriviamo child::profession selezioniamo solo i figli di tipo profession e se scriviamo solo child::text() selezioniamo solo i figli di tipo testo. Se scriviamo attributi::* esso accede agli attributi. Questa asse è mancata nella trattazione precedente sugli assi.

I location step possono essere combinati in location path separati da slash, ad esempio:

```
/descendant::L/child::*/child::*
```

Lo slash iniziale indica che iniziamo dal nodo radice.





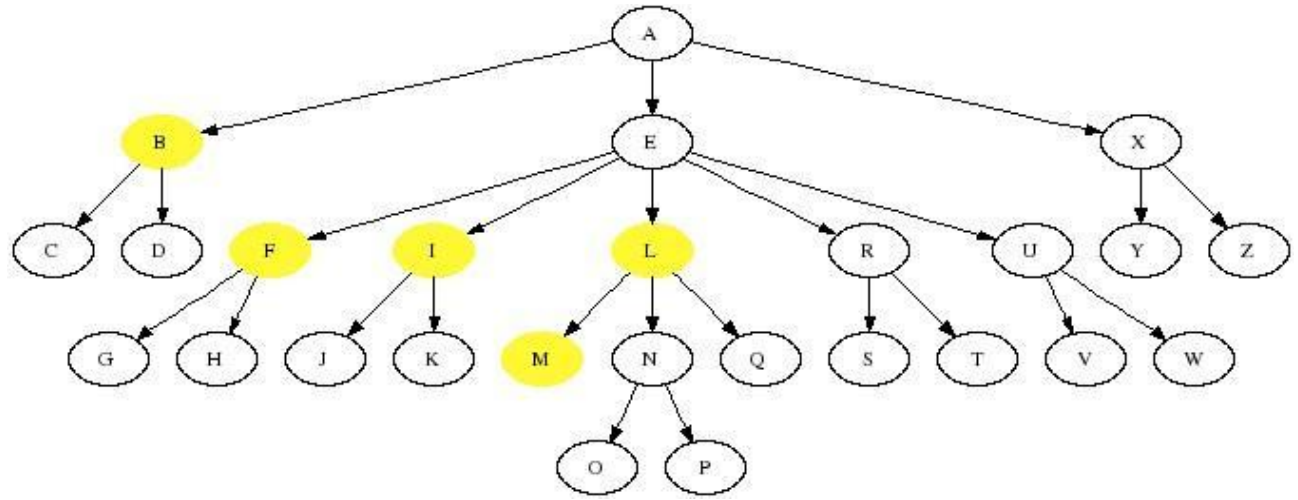
Il nostro risultato è quindi O e P. Ci manca l'ultima componente che sono i filtri. Un filtro è una comune azione booleana di un location path. Lo scopo del filtro è quello di filtrare i nomi, scegliendo quelli che soddisfano alcune consizioni. Prendiamo ad esempio:

```
/descendant::*[child::*]
```

Il risultato sarà:

Perché il primo step mi seleziona tutti i discendenti della radice, a questo punto il filtro dice di selezionare tutti quelli che hanno almeno un figlio [child::*]. Vediamo un altro esempio:

```
/descendant::*[following-sibling::*/following-sibling::*]
```



Sostanzialmente si prendono tutti i nodi che hanno almeno due nodi a destra. Si può infine correggere con i connettivi booleani. Ad esempio

```
/descendant::*[not(child::*)]
```

Prendiamo i nodi che non hanno un figlio, cioè le foglie. Oppure con:

```
/descendant::*[child::* and preceding-sibling::*]
```

Prendiamo i nodi che hanno un figlio e almeno un fratello a sinistra, come il nodo R. E se:

```
/descendant::*[ancestor::B or preceding::B]
```

Prendiamo i nodi che seguono B nell'ordine del documento. Quindi un nodo viene selezionato se fa parte della sua discendenza o se è un fratello di destra, quindi tutti i nodi da C fino a Z.
Ora si vede l'esempio il cui file è nella cartella riguardante tale capitolo.

Ora andremo ad analizzare XQuery. L'input di ogni interrogazione è una sequenza dove ogni elemento della sequenza o sono elementi XML oppure valori atopici come stringhe, ecc. Il legame tra XPath e XQuery è che XPath è una parte di XQuery, cioè XQuery usa XPath. Tipicamente il processo di un XQuery è il seguente:

1. Carico uno o più documenti XML che voglio interrogare
2. Uso XPath per recuperare sequenze di nodi da questi documenti
3. Uso delle primitive, cioè delle clausole specifiche di XQuery per elaborare questo insieme di nodi
4. Costruisco un risultato
5. Restituisco il risultato della mia interrogazione

Alcuni di questi passi sono facoltativi e si può vede XQuery come un linguaggio di programmazione su sequenze, quindi è possibile scrivere qualsiasi funzione computabile. Le interrogazioni in XQuery si chiamano **FLWOR expression**, il cui nome è acronimo delle clausole che definiscono l'interrogazione che sono al pari di select, group by, ecc. di SQL. Vediamo le clausole nello specifico:

- La clausola **for** associa una o più variabili ad una espressione XQuery, ricordando che il risultato di una espressione è una sequenza di nodi. For itera sul risultato di questa sequenza, proprio come il for nel linguaggio di programmazione
- La clausola **let** assegna sempre una variabile ad una espressione, però con una semantica diversa, perché valuta l'espressione e assegna il risultato ad una variabile, non va in modo iterativo. Alla fine di queste due clausole l'assegnamento variabile e valore prende il nome di **tupla**. A questo punto le prossimo clausole servono per far qualcosa con queste tuple.
- La clausola **where**
- Clausola **order by**
- Clausola **return**, per restituire un risultato

Le uniche obbligatorie sono una tra for e let e return, le altre sono opzionali. Vediamo tale linguaggio tramite esempio. Sullo stesso file dell'esercizio precedente.

```
for $act in //ACT
return <act>
         {$act/TITLE}
         <count>{count($act/SCENE)}</count>
       </act>
```

Le parentesi grafe indicano che vogliamo quel risultato nell'output. Le variabili sono sempre prefissati dal dollaro. IL primo elemento richiesto quindi è il titolo degli atti. <count>{count($act/SCENE)}</count> restituisce nivee il numero di scene in tali atti. La prossima query è equivalente:

```
for $act in //ACT
let $scene := $act/SCENE
return <act>
         {$act/TITLE}
         <count>{count($scene)}</count>
       </act>
```

Si noti il := per l'assegnazione e l'in per l'iterazione del for. Ora vogliamo tutti i titoli e il conto degli speach per gli atti che hanno più di 100 speach:

```
for $act in //ACT
```

```
let $speech := $act//SPEECH
where count($speech) > 100
return <act>
```

```
        {$act/TITLE}
        <count>{count($speech)}</count>
      </act>
```

Vediamo ora tutti i titoli e le battute (speach) ordinate in ordine decrescente rispetto al numero di battute:

```
for $act in //ACT
let $line := $act//LINE
order by -count($line)
return <act>
        {$act/TITLE}
        <count>{count($line)}</count>
      </act>
```

Si noti che il risultato non è esattamente un documento XML, pur avendo comunque gli elementi con medesima definizione di validità. Vediamo le ultime due. Per commentare XQuery si faccia il sorriso. Vogliamo ora trovare l'atto con il massimo numero di battute:

```
let $count := for $act in //ACT
              return count($act//LINE)
let $max := max($count)
for $act in //ACT
where count($act//LINE) = $max
return <act lines="{$max}">{$act/TITLE}</act>
```

La prossima è simile alla precedente:

```
let $act := for $a in //ACT
            order by -count($a//LINE)
            return $a
return $act[1]/TITLE
```

La differenza è che la prima query restituisce anche un numero maggiore di due atti se hanno medesima lunghezza, mentre la seconda solo uno. Infine XQuery ha una funzione **fulltext**, cioè fa delle interrogazioni orientate al testo, ad esempio tutte le battute che contengono la parola *hath*:

```
//LINE[ . contains text "hath" ]
```

Se lo scrivessi in XPath avrei tutte quelle contengono la stringa *hath* e non la parola:

```
//LINE[contains(., "hath")]
```

C'è una differenza perché quando utilizziamo una query di tipo fulltext all'inizio c'è un processo che si chiama **tokenziation** in cui andiamo a prendere le parole e il testo e faccio un certo numero di trasformazioni. Ad esempio si mette tutto in minuscolo, si tolgono gli accenti, ed altre. L'effetto è che la prima query è esattamente uguale la prossima query:

```
//LINE[ . contains text "...Hath!" ]
```

Sono uguali appunto perché vengono fatte tali trasformazioni all'inizio. Posso combinare tutte le parole che contengono diverse parole (si veda il file dell'esercizio).

Per analizzare la rilevanza della parola, cioè un fattore che ha a che fare con il numero di volte in cui si presenta tale parola:

```
for $hit score $score in //LINE[ . contains text "love" ]
order by $score descending
return <hit score='{format-number($score, "0.00")}'>{$hit}</hit>
```

Si guardi attentamente l'esercizio, noi concludiamo qui tale trattazione perché non abbiamo il tempo di analizzare altro.

Una query che si ha saltato prima è la validazione, è possibile anche validare documenti rispetto a schemi XML, si mette il documento in una variabile doc, lo schema in una variabile schema e applico tale query:

```
# XML document in dream.xml
let $doc := doc('dream.xml')
let $schema := 'play.dtd'
return validate:dtd($doc, $schema)

# current XML database
let $schema := 'play.dtd'
return validate:dtd(., $schema)
```

Se facciamo delle modifiche al documento XML dopo aver importato il DB in basex tale modifica non si presenta, c'è tutta una parte di XQuery per modificare i documenti XML.

A questo punto si è svolto l'esercizio XML makes R. In R si può eseguire XPath, ma non XQuery.

6. Text Mining

Siamo giunti ad analizzare dei dati che sono il meno strutturati possibile. Si chiamano dati strutturati o regolari dei dati che possono essere messi in un DFrame. Per quanto riguarda il Text Mining parleremo inizialmente di tale argomento in maniera teorica e successivamente seguiremo il libro Text Minig, il quale si propone di utilizzare l'idea del Tidy per analizzare dati testuali.

6.1 Espressioni regolari e automi

Iniziamo con una trattazione più teorica dalle dispense di Dovier e Giacobazzi che si trovano sul sito.

6.1.1 Automi a stati finiti

L?obiettivo è quello di introdurre due formalismi per descrivere insiemi finiti di parole. Una parola è una sequenza di caratteri presi da un certo alfato. Definiamo alfabeto un insieme di simboli. Vorremmo caratterizzante un insieme finito insieme inviti di parole. Un alfabeto potrebbe essere {0,1} e 01101 è una parola. Se prendessimo i numeri divisibili per 5, essi sono infiniti e quindi vogliamo dare una caratterizzazione finita. Per alcuni linguaggi, chiamati regolari, è possibile dare una caratterizzazione finita in termini di automi e espressioni regolari. Non sempre è possibile trovare una espressione regolare per un linguaggio.

Conviene fissare a questo punto un po' di sintassi. Un simbolo è un elemento del nostro alfabeto, uno stringa o parola è una sequenza di simboli giustapposti. Esiste una parola vuota che indiciamo con :

$$|\epsilon| = 0$$

- $a_1 \ldots a_j$, con j $\in$ {1,...,n} è detta un **prefisso** di w;
- $a_i \ldots a_n$, con i $\in$ {1,...,n} è detta un **suffisso** di w;
- $a_i \ldots a_j$, con i,j $\in$ {1,...,n}, i $\leq$ j, `e detta una **sottostringa** di w;
- è sia prefisso che suffisso che sottostringa di w.

Una operazione importante è la concatenazione di due stringhe, se abbiamo due stringhe v e w, possiamo concatenarle e creare una nuova stringa z, $z = v\,w$. Si noti che $(a\,b)c = a\,(b\,c)$.

Un alfabeto è un insieme finito di simboli. Un linguaggio formale è un insieme di parole formate su un determinato alfabeto. Un linguaggio potrebbe essere finito, casistica poco interessante perché ci interessa trovare caratteri finiti in casi infiniti.

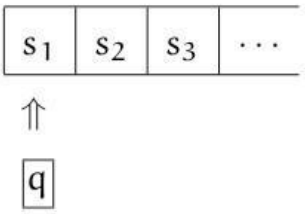
Un **alfabeto** Σ è un insieme finito di simboli. Un **linguaggio formale** (in breve linguaggio) è un insieme di stringhe di simboli da un alfabeto Σ. L'insieme vuoto $\varnothing$ e l'insieme $\{\varepsilon\}$ sono due linguaggi formali di qualunque alfabeto. Con Σ^* verrà denotato il linguaggio costituito da tutte le stringhe su un fissato alfabeto Σ. Dunque

$$\Sigma^* = \{a_1 \ldots a_n : n \geq 0, a_i \in \Sigma\}$$

Ad esempio, se $\Sigma = \{0\}$, allora $\Sigma^* = \{\varepsilon,0,00,000,...\}$; se $\Sigma = \{0,1\}$, allora $\Sigma* = \{\varepsilon,0,1,00,01,10,11,000,...\}$.

Se il nostro alfabeto è =,1 allora il nostro linguaggio Σ^* sarà tutte le parole che posso scrivere con questi due simboli.

Veniamo quindi al concetto di **automa**. Un automa è fondamentalmente un modello di calcolo a stati fini, cioè si può trovare ad un numero finito di stati. Un esempio è l'ascensore. Un ascensore di ricorda il piano in cui si trova (stato corrente) e quello che l'automa leggerà sarà la frequenza di tali che si possono pigiare. Si può rappresentare mediante una sorte di macchina che ha una memoria, un unico registro di memoria che contiene lo stato e legge una sequenza di simboli, cioè una parola finita.



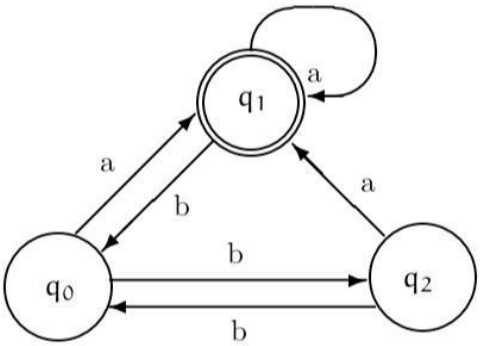
Questo automa a partire dallo stato in cui si trova l'automa legge un simbolo e cambia eventualmente lo stato e poi la tesina si sposta di uno in avanti, cioè legge il simbolo successivo. Dobbiamo immaginarlo come una sorta di macchina che ha un unico registro di memoria e in base allo stato in cui si trova e il simbolo che trova decide di andare in un altro stato. É importante notare che l'automa può spostarsi in un numero finito di simboli e può leggere solo un simbolo alla volta.

Il movimento dell'automa posso rappresentarlo tramite un grafo o una matrice di transizione, che non ha nulla a che fare con la matrice di adiacenza.

La matrice di transizione ha sulle righe gli stati e sulle colonne i simboli.

	a	b
q_0	q_1	q_2
q_1	q_1	q_0
q_2	q_1	q_0

L'immagine dice che se cono nello stato q_0 nello stato successivo sarà nel simbolo .



I nodi sono gli stati e gli archi sono i simboli letti. Quindi il comportamento dell'automa può essere identificato in questi due modi. I nodi cerchiati sono quelli finali. La differenza con il formalismo della macchina di Turing (marchingegno che legge dei simboli su un nastro e a partire dagli stati produce degli output) sono:

1. L'automa a stati finiti non crea output
2. L'automa può andare solo avanti e non può tornare indietro, non può rivedere quale sia la sua storia. Tutta la storia viene codificata nello stato attuale.

Se prendiamo gli automi e aggiungiamo questi due proprietà ottimo il formalismo più espressivo che si chiama Macchina di Turing. Ha un formalismo molto importante perché rappresenta il formalismo di ogni calcolatore di qualsiasi grandezza. Possiamo immaginar allora il nostro computer come una testina che ha un output e può andar avanti e indietro.

Proviamo a dare una definizione più formale

6.1.2 Automi deterministici

Un automa a stati finiti deterministico (DFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F\rangle$ dove:

- Q è un insieme finito di stati;
- Σ è un alfabeto (alfabeto di input);
- $\delta: Q \times \Sigma \longrightarrow Q$ è la funzione di transizione;
- q_0 è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati finali.

Quindi per ogni stato e per ogni simbolo l'automa deve sapere che cosa fare. Quindi per ogni nodo ci deve essere un numero di archi pari al numero di simboli che escono da quel nodo. Gli stati finali corrispondono agli stati che servono per decidere il linguaggio che l'automa riconosce.

Per definire il linguaggio accettato dal nostro automa intuitivamente faccio così: dobbiamo decidere se una parola appartiene al linguaggio, con l'automa leggo la parola e se arrivo allo stato finale allora la parola è accettata.

Useremo p, q, r con o senza pedici per denotare stati, P, Q, R, S per insiemi di stati, a, b con o senza pedici per denotare simboli di Σ, x, y, z, u, v, w sempre con o senza pedici per denotare stringhe.

Dalla funzione δ si ottiene in modo univoco la funzione $\delta : Q\ x\ \Sigma^* : -> Q$ nel modo seguente:

$$\delta(q, \epsilon) = q$$
$$\{\delta(q, w\ a) = \delta(\delta(q\ .\ w), a)$$

Una stringa x è detta essere accettata da un DFA $M = \langle Q, \Sigma, \delta, q0, F\rangle$ se δ (q0, x) $\in$ F. Il linguaggio accettato da M, denotato come L(M) è l'insieme delle stringhe accettate, ovvero:

$L(M) = \{x \in \Sigma^* : \delta(q_0, x) \in F\}$.

Un linguaggio L è detto regolare se è accettato da qualche DFA, ovvero se esiste M tale che L = L(M).

Quindi un linguaggio è detto regolare se esiste qualche automa a stati finiti che lo accetta.

ESEMPIO 3.5. $\varnothing$ e Σ^* sono linguaggi regolari. Sia $\Sigma = \{s_1, \ldots, s_n\}$: un automa M_0 che riconosce il linguaggio $\varnothing$ (ovvero: nessuna stringa è accettata) è il seguente:

	s_1	...	s_n
q_0	q_0	...	q_0

ove $F = \varnothing$. Infatti, poiché $\forall x\,(x \notin \varnothing)$, si ha che:

$$(\forall x \in \Sigma^*)(\hat{\delta}(q_0, x) \notin F).$$

Un automa per Σ^*, è invece l'automa M_1:

	s_1	...	s_n
q_0	q_0	...	q_0

ove $F = \{q_0\}$. Si dimostra facilmente infatti, per induzione su $|x|$ che

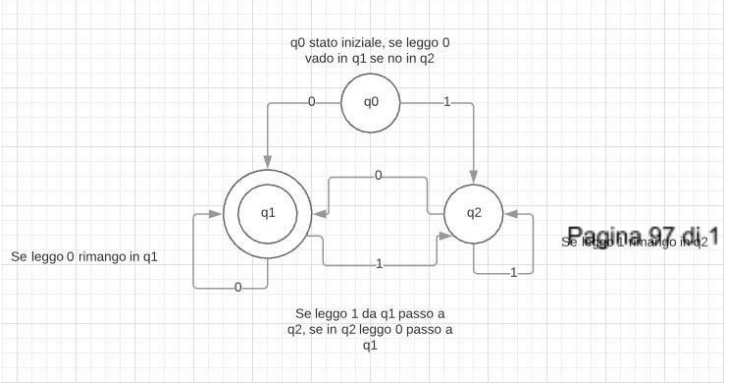
$$(\forall x \in \Sigma^*)(\hat{\delta}(q_0, x) = q_0).$$

Facciamo due semplici esempi. Proviamo a creare un automa che riconosce il linguaggio vuoto, che sicuramente è un linguaggio perché sotto insieme di Σ* , allora basterà mettere l'insieme degli stati finali uguale a vuoto. Questo automa è quello scettico a cui non va bene niente.

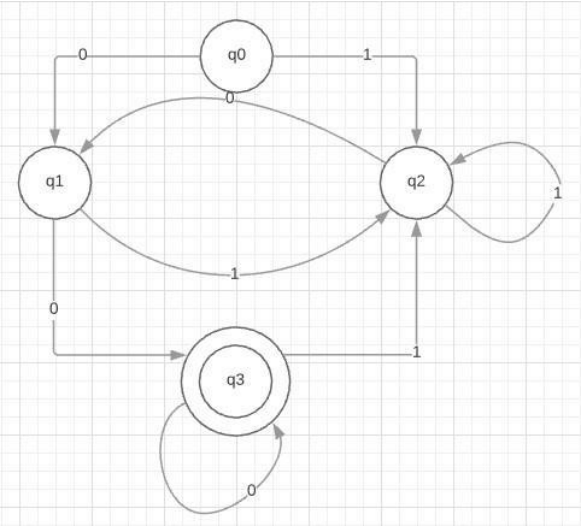
Il suo speculare è l'automa a cui gli va bene tutto, il cosiddetto credulone.

Prendo il linguaggio {0,1}:

- Automa per numeri divisibili per due



• Automa che riconosce i numeri divisibili per 4.



6.1.3 Automi non deterministici

Un'altro formalismo è basta sugli automi non deterministici. Questi automi posson opercorrere più strade parallelamente. Quindi se si trova su uno stato e legge un qualche simbolo può andare su più di uno stato o in nessuno.

Un automa a stati finiti non-deterministico (NFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F\rangle$ dove Q, Σ, q_0 e $F \subseteq Q$ mantengono il significato visto per gli automi deterministici, mentre la funzione di transizione δ è ora definita:

$$\delta : Q \times \Sigma \longrightarrow \wp(Q) .$$

Si osservi che è ora ammesso: $\delta(q,a) = \varnothing$ per qualche $q \in Q$ ed $a \in \Sigma$. Anche per gli NFA dalla funzione δ si ottiene in modo univoco la funzione δ: $Q\times\Sigma^* \to \wp(Q)$ nel modo seguente:

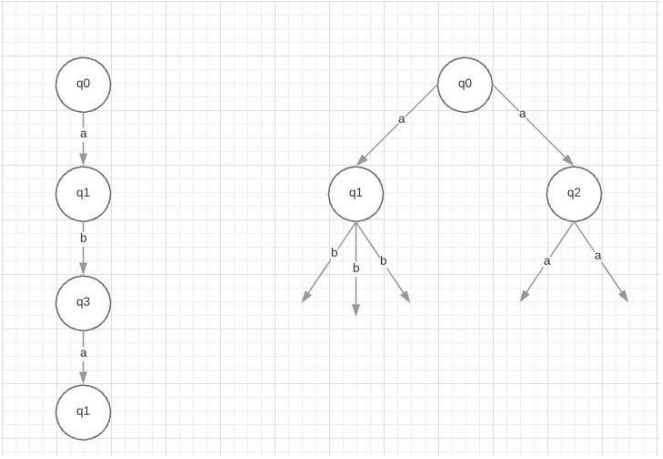
$\delta(q, \epsilon) = \{q\}$

$\{\delta(q, w\ a) = \cup_{p\ \ in\ \delta(q,w)}\ \delta(p, a)$

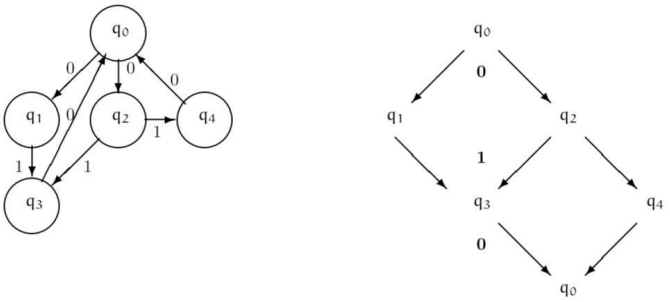
Una stringa x è accettata da un NFA $M = \langle Q, \Sigma, \delta, q0, F\rangle$ se $\delta(q_0, x) \cap F \neq \varnothing$. Il linguaggio accettato da M è l'insieme delle stringhe accettate, ovvero:

$$L(M)=\{x\in\Sigma^* : \delta(q_0,x)\cap F\neq\varnothing\}.$$

A questo puto data una parola il mio automa non risponderà più con un unico stato ma con più stati . Per vedere se la parola è accetta se almeno una delle computazioni è andata a buon fine. Deve esistere almeno un ostato che ho raggiunto che non è vuoto. Un automa deterministico, semplificando, può esser visto come un cammino. Nel caso non deterministico abbiamo un albero.



Apparentemente questo è un formalismo molto più potente. In verità non lo è. Perché è possibile dimostrare un teorema di equivalenza. I linguaggi accettati da un automa deterministico e non deterministico sono esattamente uguali. Se abbiamo un linguaggio accetto da un automa non deterministico sarà accettata da quello deterministico e viceversa visto che quello deterministico è sotto caso dell'automa non deterministico.



Prima di vedere il teorema inverso, ragioniamo sull'esempio di Figura 1.

Proviamo a seguire la computazione dell'automa sulla stringa 010. All'inizio la computazione si biforca in modo non deterministico sui due stati q_1 e q_2. Ciò può erroneamente far pensare che sia necessaria una struttura dati ad albero per rappresentare una computazione non-deterministica. Al secondo livello, quando il carattere 1 è analizzato, sia da q_1 che da q_2 si raggiunge lo stato q_3. Non `e necessario ripetere lo stato in due nodi distinti. Inoltre lo stato q_4 è pure raggiungibile da q_2. Proseguendo, si vede che le due computazioni non deterministiche confluiscono nello stato q_0.

I passaggio da non deterministico e deterministico determina una esplosione esponenziale del numero degli stati.

6.1.4 Automi -transizioni

L'ultimo formalismo sono gli automi con -transizioni. In questo caso l''idea è che possono cambiare lo stato anche se non leggo alcun simbolo.

Un NFA con ε-transizioni è una quintupla $\langle Q, \Sigma, \delta, q_0, F\rangle$ dove: Q, Σ, q_0 e $F \subseteq Q$ sono come per gli automi non deterministici, mentre la funzione di transizione δ `e ora definita

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \wp(Q) .$$

L'idea è che da uno stato è permesso passare ad un altro stato anche senza "leggere" caratteri di input.

La costruzione della funzione $\delta : Q \times \Sigma^* \rightarrow \wp(Q)$ nel caso dei ε-NFA risulta leggermente più complessa che nei casi precedenti. Per far ciò si introduce la funzione ε-closure che, applicata ad uno stato, restituisce l'insieme degli stati raggiungibili da esso (compreso sé stesso) mediante ε-transizioni. La costruzione di tale funzione è equivalente a quella che permette di conoscere i nodi raggiungibili da un nodo in un grafo e può facilmente essere calcolata a partire dalla funzione δ (un arco $p \rightarrow q$ si ha quando $q \in \delta(p, \varepsilon)$). Il concetto di ε-closure si estende in modo intuitivo ad insiemi di stati:

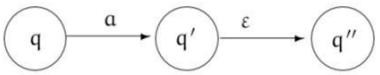
$$-closure(P) = \bigcup_{p \in P} -closure(p)$$

δ si può ora definire nel modo seguente:

$\delta(q, \epsilon) = \epsilon\text{-}cl\ o\ s\ u\ r\ e\ (q)$

$\{\delta(q, w\ a) = \cup_{p\ in\ \delta(q,w)}\ \epsilon\text{-}cl\ o\ s\ u\ r\ e\ (d\ el\ t\ a\ (\ p, a))$

Si noti che in questo caso δ(q,a) può essere diverso da δ(q,a). Ad esempio, nell'automa:



Si ha che δ(q,a) = {q′}, mentre δ(q,a) = $\cup_{p\ in\ \delta(q,w)}$ $\epsilon\text{-}cl\ o\ s\ u\ r\ e\ (d\ el\ t\ a\ (\ p, a)) = \{q', q''\}$.

Definiamo dunque il linguaggio accettato dall'automa L(M) = $\{x \in \Sigma^* : \delta(q_0, x) \cap F \neq \varnothing\}$.

Si osservi come per questa classe di linguaggi si potrebbe assumere che l'insieme F abbia esattamente un elemento. Si osservi inoltre che δ(q, x) = ε-closure(δ(q, x)).

Anche questa cosa non aggiunge espressività, esso è equivalente ad un automa non deterministico e quindi anche un automa deterministico.

6.1.5 Espressioni Regolari

Una ER è una espressione in una algebra dove gli elementi sono insieme di parole su un alfabeto e le operazioni sono operazioni su insiemi: unione, concatenazione e chiusura. Ogni volta che applico una operazione ho un elemento del mio dominio.

L'unione viene scritta così: $L_1 + L_2$, far attenzione che tipo in R si scrive $L_1 \mid L_2$. La chiusura quindi è una operazione unaria.

Se scriviamo ad esempio, per quanto riguarda la chiusura: (0 + 1) * sono tutte le parole che posso scrivere con i simboli 0 e 1 con la parola vuota .

Sia Σ un alfabeto. Le espressioni regolari su Σ e gli insiemi che esse denotano sono definiti ricorsivamente nel modo seguente:

(1) ∅ è una espressione regolare che denota l'insieme vuoto.
(2) ε è una espressione regolare che denota l'insieme {ε}.
(3) Per ogni simbolo a ∈ Σ, a è una espressione regolare che denota l'insieme {a}.
(4) Se r e s sono espressioni regolari denotanti rispettivamente gli insiemi R ed S, allora (r + s), (rs), e (r*) sono espressioni regolari che denotano gli insiemi R ∪ S, RS, e R^* rispettivamente.

Se r è una espressione regolare, indicheremo con L(r) il linguaggio denotato da r. Tra espressioni regolari valgono delle uguaglianze che permettono la loro manipolazione algebrica. Varrà che r = s se e solo se L(r) = L(s).

Esercizio: proviamo a scrivere le espressioni regolari degli automi di prima:

- r_2 = (0 + 1) * 0
- r_4 = (0 + 1) * 00

Esiste un teorema di equivalenza (McNaughton & Yamada, 1960). Sia r una espressione regolare. Allora esiste un ε-NFA M tale che L(M) = L(r).

Quindi le espressioni regolari sono equivalenti agli automi. Sia dia una occhiata alla dimostrazione.

Vediamo ora il teorema inverso: Sia M un DFA. Allora esiste una espressione regolare r tale che L(M) = L(r).

Si dia una occhiata alla dimostrazione.

Si vede ora brevemente le espressioni regolari in R capitolo 14 di R for DataScience.

6.2 Text Mining

Una cosa che possiamo fare con un testo è contare le parole, cioè la loro frequenza. Oppure potremmo analizzare il sentimento dello scrittore del testo (es. rabbia, felice, ecc.) tramite la sentiment analisys. Altre cose che possiamo fare è se abbiamo un corpus di documenti (collezione di documenti) possiamo vedere per ognuno di questi documenti quali sono le parole più rilevanti, non solo tramite la frequenza, ma anche con la tecnica più avanzata tf-idf, che cerca di estrapolare le parole che sono caratterizzante di quel documento. Se no possiamo anche analizzare gli n-grammi cioè sequenze di parole accostate, ad esempio: se abbiamo due parole W_1 e W_2 ci interessa l'evento in cui le parole sono vicine, quindi ci interessa una relazione tra queste due parole, piuttosto che parole che sono nel medesimo paragrafo o sezione. Inifne vedremo come suddividere il nostro corpus in cluster che corrispondono a delle comunità coese, cioè che parlano di un certo topi con la **topic modelling**, il quale non suddivide in maniera netta i documenti, ma assegna una probabilità di appartenenza al topico (ad esempio 80% politica e 20% finanza) con un approccio non supervisionato. Le seguenti trattazioni sono svolte nel libro.

7. Big Data and Internet of Things

Accenneremo alcune tecnologie per analizzare i big data. Innanzitutto dobbiamo capire cosa sono i BD. IL corso si è comunque fondato sull'analisi dei piccoli dati. Se abbiamo dati dell'ordine dei byte si possono usare i metodi fino ad ora utilizzati, dati dell'ordine dei terabyte possono essere analizzati con SQL e la teoria relazionale. Per dati più grandi, dell'ordine dei petabyte 10^{15} byte, non basta neanche il modello relazionale. In questo caso per cui c'è un problema di tempo e di spazio. In generale ci sono dei casi in cui l'odiane dei dati a oltre a quello che possiamo memorizzare, fondamentalmente dobbiamo trovare forse alternative per archiviare e fare le analisi.

Per giungere al concetto di BD siamo passati per 4 fasi:

1. Nascita del **web** (1989), che permette a tutti o quasi a tutti di crearsi la pagina web e di mettere in rete una gran quantità di dati di tutti i tipi.
2. l'arrivo dei **social network** (FB nel 2004, twitter 2006), anche questa è stata una spinta ulteriore a mettere dati sul sistema.
3. L'introduzione degli **smartphone** (prima generazione dell'iPhone nel 2007). Il fatto di non avere più semplicemente dei pc sulla scrivania, ma anche in tasca ha facilitato l'every where e l'every time della condivisione.
4. L'avvento dell'**internet delle cose** (Internet of Things), cioè l'introduzione nelle cose di piccoli chip che hanno dei sensori e sono connessi in rete e immettono continuamente dati nella rete. Si pensi ad esempio alla domotica, in cui alcuni oggetti (tapparelle, caldaia, ecc.) possono essere comandati tramite smartphone e cose del genere. Stiamo andando verso un modo in cui non solo le pagine web saranno connesse tra loro ma anche gli oggetti e questo causerà un ennesimo salto verso la condivisione dei dati.

I big data sono un dataset le cui dimensioni sono maggiori di quelle di un "normale" database (report sui big data di MaKinsey). I dati sono di fatto un'attività indispensabile per le aziende moderne e sopratutto c'è una forte spinta nell'utilizzare i BD anche nel settore pubblico per migliorare la vita e l'economie nazionali.

Fondamentalmente i BD sono caratterizzate dalle tre V:

- **Volume**: si parla di BD quando i dati sono sufficientemente ampi da non esser rappresentate in una base di dati
- **Velocità**: i dati arrivano molto velocemente, quindi il tempo di elaborazione deve essere molto basso.
- **Varietà**: i dati sono spesso molto vari, cioè incidono dati strutturati, semistrutturati e non strutturati.
    - Strutturati sono quelli che sono definiti in un modello come quello relazionale
    - Non strutturati, sono dati che non sono parte di un modello (testo, audio, immagini, video, …)
    - Semistrutturati, sostanzialmente del testo etichettato da dei tag.

CI sono anche dei problemi associati a tale proliferazione dei dati:

- **Privacy**: per esempio i dati relativi alla cartella clinica o al conto corrente sono privati e quindi si vogliono proteggere, d'altra parte sono proprio i dati sensibili che possono essere più utili alla persona stessa. Se la cartella clinica fosse monitorata e analizzata chiaramente questo sarebbe un grosso vantaggio per tutti. Quindi c'è un trade-off tra privacy e utilità
- **Data security**: vogliamo proteggere i dati da intrusioni altrui, quindi è un argomento legato alla privacy. Anche se si ha deciso di cedere i dati, si vuole che essi vengano protetti. Se cediamo le informazioni all'ospedale non vogliamo che l'ospedale li condivida.
- **Legal issues**: se ci si pensa un dataset può essere copiato immediatamente. Quindi la possibilità di duplicare n DSet è enorme e in più persone possono lavorare su un DSet, cosa che non avviene con altri elementi (un quadro, ecc. ad esempio se ho una copia della gioconda essa è unica e non riproducibile). Se si ha un DSet come si fa a dire che il DSet è di qualcuno? Bisognerebbe avere dei diritti sul DSet e di conseguenza un modo onesto per usare i dati è citare la fonte o non modificare il DSet. C'è tutta una serie di questioni legali che sono legate al fatto che un dato può essere replicato con grande facilità e può essere acceduto da più persone contemporaneamente.

7.1 Small Data First

È bene affrontare prima i dati piccoli e poi quelli grossi. Ci son due buoni motivi per cui i nostri BD sono dei small data camuffati:

1. Potrebbe essere che abbiamo un DSet molto ampio, ma ci bastano pochi dati di questo insieme di dati per fare la nostra analisi. Quindi possiamo estrapolare i dati necessari per la nostra challenge. QUinid possiamo trovare un campione o un sottoinsieme
2. Oppure il nostro DSet può essere effettivamente grande e non può essere compresso in nessun modo, ma può essere suddiviso in tanti piccoli pezzi e ogni pezzo può essere analizzato separatamente. Quindi se abbiamo un DSet di 10 terabyte lo si divide in 100 parti, quindi ogni parte contiene 100 gb e questi possono essere utilizzati sul disco fisso e quindi possiamo archiviarlo in una base di dati. Questa tecnica è chiamata **map reduce** o **split apply and combine**. Se il nostro obiettivo è calcolare il massimo, si divide il BD in gruppi, si calcola il massimo locale e poi il massimo complessivo. Non sempre è fattibile ovviamente questa procedura, dipende se la procedura è parallelizzabile. Un esempio di operazione non parallelizzabile è la mediana.

Vedremo tre approcci, la prima cosa che vedremo sono delle tecniche per rendere efficiente il nostro codice R, perché quando lavoriamo con bigiata abbiamo problemi di tempo. Per questo gli algoritmi usati devono essere molto efficienti. Quindi la cosa più difficile è aumentare l'intelligenza e non la memoria.

Alcune volte il nostro problema si può spezzettare (fase di split) e ogni pezzo può essere analizzato e alla fine tutto può esser messo assieme. Vedremo questo approccio con pacchetti che lavorano su matrici. Per analizzare ciò useremo Datacamp.

Guardare cartella su desktop "writingefficientRcode" e "scalable". Quando si ha una macchina performante, uno spazio sufficiente e degli algoritmi efficienti quello che dobbiamo fare è sostanzialmente usare la tecnica split, divide and combine. Questa è la strategia più comune per affrontare la problematica del BigData.

7.1.1 BlockChain

Una delle reti più famose è la blockchain, che è un bel esempio di base di dati distribuita, cioè che usa la computazione distribuita presentata ieri. Quindi è il legame tra i DB relazionali e quelli distribuiti. La BC in più utilizza metodi crittografici per rendere sicure le trattazioni. La BC è nata con le cripto valute. Il problema è quello della double spending, cioè che la moneta si può utilizzare due volte. Essa è una tecnologia interessante perché mette assieme molte cose. Vediamo ora alcuni video e poi una semplice implementazione in R. Si legga il documento "Building a blockchain in R", procediamo com alcuni commenti di tale lavoro.

Il codice HASH in questo caso è crittografico, cioè particolari HASH utili nell'ambito della crittografia. Servono per: dato un insieme di dati di dimensione arbitraria crearne un riassunto (digest). Lo possiamo immagine come impronta digitale di un blocco. Quindi è possibile dare in pasto un blocco visto come una stringa e questa funzione HASH ci da un codice di 256 bit (nel nostro caso). La particolarità di questa funzione è che è molto facile da calcolare, ma la complessità della funzione inversa è molto alta. Quindi dato l'HASH trovare il blocco è molto difficile. L'unico approccio per calcolare l'inversa è un approccio di forza bruta, per cui l'unico approccio sensato è un po' come avere un bancomat e provare tutte le combinazioni del codice. Quindi esiste una libreria che implementa gli HASH crittografici.

```
#  Creates hash digests of arbitrary  R  objects library(digest)

#  add hash value to  the  current  block hash_block <-
function(block){
    block$new_hash <- digest(c(block$index, block$timestamp,
                                            block$data,
                                            block$previous_hash), "sha256")
    return(block)
}
```

SI noti che l'HASH dipende dallo HASH di quello precedente, che dipende da quello precedente e così via. Queste funzioni non sono continue, se cambia un byte dell'informazione cambia di molto lo HASH.

Calcolare un HASH non è troppo difficile. Interviene ora il proof-of-work. L'introduzione di un nuovo blocco in una catena deve essere qualcosa di difficile e costosa, così come la modifica, se no potremmo farlo

in maniera facile. Nelle criptovalute il blocco è la valuta, quindi se fosse semplice crearli o modificarle perderebbero di valore. Il PFW è un algoritmo che controlla la difficoltà nel creare un nuovo blocco. Questo algoritmo ovviamente tiene conto anche dell'evoluzione delle macchine. Troviamo una implementazione di questa funzione che non fa altro che cercare, dato un numero in input, il primo numero che sia divisibile per quel numero e anche per 99. Trovare questo numero non è facile. Trovare il primo numero che sia divisibile per entrambi non è facile. Il tempo di questo codice cresce in modo esponenziale con i blocchi che devo andare a modificare.

```
### Simple Proof of Work algorithm proof_of_work <-
function(last_proof){
    proof <- last_proof + 1

    # Increment the proof number until a number is found that is divisable by 99 and by the proof of the previous block
    while (!(proof %% 99 == 0 & proof %% last_proof == 0 )){
            proof <- proof + 1
    }

    return(proof)
}
```

```
proof0 = 1
for (i in 1:10) { cat(paste0(proof0, "\n"))
    proof1 = proof_of_work(proof0) proof0 =
    proof1
}
```

```
## 1
## 99
## 198
## 396
## 792
## 1584
## 3168
## 6336
## 12672
## 25344
```

Questo ovviamente non è il PFW originale. Dato un blocco l problema è trovare la codifica HASH che ha n certo numero di digit iniziali pari a zero. Questo problema ha una complessità esponenziale in funzione della difficulty che è il numero di numeri zero all'inizio dell'HASH. Questo appena descritto è il vero PFW.

Una volta definito HASH e PFW vediamo le funzioni che generano un nuovo blocco. Quindi:

```
gen_new_block <- function(previous_block){

    #Create new Block
    new_block <- list(index = previous_block$index + 1, timestamp =
                                Sys.time(),
                                data = paste0("this is block ", previous_block$index + 1), previous_hash =
                                previous_block$new_hash,
                                proof = proof_of_work(previous_block$proof))

    # Add the hash of the current block new_block_hashed <-
    hash_block(new_block)

    return(new_block_hashed)
}
```

Affinché un blocco possa essere messo nella rete il 50% +1 degli altri blocchi deve accordare la sua validità. Quindi la blockchain può fallire se c'è una se si convince il 50% + 1 degli altri. Esiste un blocco che genera tutto:

```
# Define Genesis Block (index 1 and arbitrary previous hash)
block_genesis <- list(index = 1,
                      timestamp = Sys.time(), data =
                      "Genesis Block", previous_hash =
                      "0", proof = 1)

block_genesis <- hash_block(block_genesis)
```

A questo punto possiamo creare la nostra blockchain:

```
# First block is the genesis block blockchain <-
list(block_genesis) previous_block <-
block_genesis

# How many blocks should we add to the chain after the genesis block num_of_blocks_to_add <-
5

# Add blocks to the chain
for (i in 1: num_of_blocks_to_add){
  block_to_add <- gen_new_block(previous_block)
  blockchain[[i+1]] <- block_to_add previous_block <-
  block_to_add
}

blockchain
```

```
## [[1]]
## [[1]]$index
## [1] 1
##
## [[1]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[1]]$data
## [1] "Genesis Block"
##
## [[1]]$previous_hash
## [1] "0"
##
## [[1]]$proof
## [1] 1
##
## [[1]]$new_hash
## [1] "d962040da67d5f1ee2b94bb240dc591ead1e7ab5879311c9d80ba08584489018"
##
##
## [[2]]
## [[2]]$index
## [1] 2
##
## [[2]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[2]]$data
## [1] "this is block 2"
##
## [[2]]$previous_hash
## [1] "d962040da67d5f1ee2b94bb240dc591ead1e7ab5879311c9d80ba08584489018"
##
## [[2]]$proof
## [1] 99
```

```
##
## [[2]]$new_hash
## [1] "417bf0d2c7e1dd8f506dd99e9228df74bc1d0c5f95946eff51e6a773c011c824"
##
##
## [[3]]
## [[3]]$index
## [1] 3
##
## [[3]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[3]]$data
## [1] "this is block 3"
##
## [[3]]$previous_hash
## [1] "417bf0d2c7e1dd8f506dd99e9228df74bc1d0c5f95946eff51e6a773c011c824"
##
## [[3]]$proof
## [1] 198
##
## [[3]]$new_hash
## [1] "b0e4b9206818518e9e17ebafe70482e1334be23e33dd0c54bb8453785d7cc0a8"
##
##
## [[4]]
## [[4]]$index
## [1] 4
##
## [[4]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[4]]$data
## [1] "this is block 4"
##
## [[4]]$previous_hash
## [1] "b0e4b9206818518e9e17ebafe70482e1334be23e33dd0c54bb8453785d7cc0a8"
##
## [[4]]$proof
## [1] 396
##
## [[4]]$new_hash
## [1] "fa7f05d6fb5e2047b3ebf0091a3ac25195c08a61459f8301f54e5c33d38adc48"
##
##
## [[5]]
## [[5]]$index
## [1] 5
##
## [[5]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[5]]$data
## [1] "this is block 5"
##
## [[5]]$previous_hash
## [1] "fa7f05d6fb5e2047b3ebf0091a3ac25195c08a61459f8301f54e5c33d38adc48"
##
## [[5]]$proof
## [1] 792
##
## [[5]]$new_hash
## [1] "f3c11c3998d1e8625d7a507720223936295e1869031316518e7a07c29fb604c8"
##
##
## [[6]]
```

```
## [[6]]$index
## [1] 6
##
## [[6]]$timestamp
## [1] "2018-05-29 10:27:30 CEST"
##
## [[6]]$data
## [1] "this is block 6"
##
## [[6]]$previous_hash
## [1] "f3c11c3998d1e8625d7a507720223936295e1869031316518e7a07c29fb604c8"
##
## [[6]]$proof
## [1] 1584
##
## [[6]]$new_hash
## [1] "830ac0053014f36eb660db80e889cad28a28f1e0df954ed7e0a31b48a5d9affb"
```

C'è un problema ecologico perché stiamo utilizzando un sacco di rete. A livello di ricerche si sta pensando a modi alternativi. Anche perché non si risolve un problema completo, cioè la soluzione non ha alcuna utilità, non alza di gradino. c'è una idea alternativa che si basa sul random che si chiama Proof-of-Stake.

C'è molto fermento nell'arte digitale. Dopo la criptovaluta è il secondo prodotto. Risulta esser modo per remunerare gli artisti anche nel mercato secondario. Cioè quando si vende un'opera d'arte quello che incasso è quello che ricevo, se l'acquirente vende nuovamente il frutto di tale vendita non entra nelle tasche dell'artista. Ciò non succede nell'ambito del blockchain, in quanto una quota va nelle tasche dell'artista.

Quindi la blockchain in sostanza è un luogo dove si possono far scambi senza il bisogno di un intermediario.

7.2 Internet of Thing

Quando abbiamo introdotto i BD c'erano 4 elementi che hanno portato a tele entità. Una di queste era l0internet delle cose, cioè una rete che connette oggetti che si parlano tra di loro. Questo è un argomento abbastanza vasto. Noi ic focalizziamo su alcune cose:

1. Processing, con cui è possibile molto facilmente creare delle visualizzazioni dinamiche e iterative, che potrebbe essere ortogonale a ggplot. Se capita di voler usare una geometria ce non esiste la possiamo realizzare con processing, perché abbiamo il controllo dei pixel. Un'altro motivo è che si interfaccia molto bene con arduino
2. Arduino è una scheda, cioè un piccolo computer che può esser programmato e attraverso un cavo seriale può trasmettere i dati ad un pc il quale li può leggere ad esempio mediante processing in tempo reale.

Qua si tocca anche la velocità della raccolta dei dati. Per fare confidenza vediamo un primo sketch:

```
int nCircles = 10;
Circle[] circles = {};

void setup() {
  size(800, 500);
  background(255);
  smooth();
  strokeWeight(1);
  fill(150, 50);
  newCircles();
}

void draw() {
  background(255);
  for (int i=0; i < circles.length; i++) {
    // move bubbles
    circles[i].moveMe();
```

```
    // draw bubbles
    circles[i].drawMe();
    // draw links
    circles[i].linkMe();
  }
}

void newCircles() {
  for (int i=0; i < nCircles; i++) {
    circles = (Circle[]) append(circles, new Circle(circles.length));
  }
}

void mouseReleased() {
  newCircles();
}



class Circle {

  int id; // identificatore univoco
  float x, y; // coordinate del centro
  float radius; // raggio
  color fillcol; // colori del contenuto
  float alpha; // trasparenza
  float xmove, ymove; // vettore di movimento (direzione e velocitÃ )

  Circle(int _id) {
    id = _id;
    x = random(width);
    y = random(height);
    radius = random(10, 100);
    fillcol = color(random(255), random(255), random(255));
    alpha = random(255);
    xmove = random(-2, 2);
    ymove = random(-2, 2);
  }

  void drawMe() {
    noStroke();
    fill(fillcol, alpha);
    ellipse(x, y, radius*2, radius*2);
  }

  void moveMe() {
    x += xmove;
    y += ymove;
    if (x > (width + radius))   x = 0 - radius;
    if (x < (0 - radius))       x = width + radius;
    if (y > (height + radius))  y = 0 - radius;
    if (y < (0 - radius))       y = height + radius;
    // bouncing
    //if ( (x > (width - radius))  | (x < radius) )  xmove *= -1;
    //if ( (y > (height - radius)) | (y < radius) )  ymove *= -1;
  }

  void linkMe() {
    for (int i = id + 1; i < circles.length; i++) {
      float dis = dist(x, y, circles[i].x, circles[i].y);
      float overlap = dis - radius - circles[i].radius;
```

```
      if (overlap < 0) {
        stroke(0);
        line(x, y, circles[i].x, circles[i].y);
        //float control = 50;
        //bezier(x, y, x+control, y+control, circles[i].x-control, circles[i].y-
control, circles[i].x, circles[i].y);
      }
    }
  }

}
```

Arduino è una piccola scheda elettronica dotata di un micro controllore e dei registri di memoria e un input sia digitali che analogici. È un processo interessante, inventato in Italia, è curioso che il nome deriva dal Bar frequentato dal professore e i suoi studenti. Quindi è una scheda che dei sensori e degli attuatori. I sensori sente delle informazioni esterne. Gli attuatori sono delle componenti che fanno qualcosa, ad esempio un Led o un motore, e così via. Quindi sono componenti che attuano qualcosa. In base a quello che la scheda sente tramite i sensori produce attraverso i suoi attuatori. Il linguaggio è chiamato Wiring.

CI sono 4 componenti:

1. L'hardware: una manopola (potenziometro) che si può girare come se fosse la manopola del volume, una fotocellula, un bottone (on/off) e un piezo che vibra in base al tono e la frequenza che riceve ed emette dei suoni.
2. Protocollo di comunicazione: arduino inizia e manda un byte a processing, il quale risponde con un byte. A questo punto inizi la relazione.
3. Firmware: è il software inclusa in una componente, ad esempio la macchina, un frigo, ecc. Cioè. È il codice che fa funzionare la parte di arduino.
4. Software: è la parte scritta in processing.

7. Collaborate end Comunication

Questa è la parte più importante perché l'esposizione è la valutazione del lavoro. Comunicare in modo efficace ed accattivante è un'arte che si può imparare.

La prima regola è l'eticità. Se così non fatto chi ascolta si sente fregato. Parafrasando Ipocrate:

- Se non si è riusciti a far qualcosa durante il progetto o si dice o non si cela questa cosa
- Quando si fanno delle analisi commissionate bisogna rispettare la privacy, quindi non esporre dati sensibili.
- Ricordarsi che ho dei obblighi come essere umano e quindi anche in questo caso avere una forma di rispetto.

Vediamo degli esempi di violazione: l'asse delle y è invertito, la scala è molto grande rispetto alla variazione che si vuole misurare, ecc. Si legga il testo di "Edward Tufte" in infographics nel sito del prof. Si cerchi di agire in modo corretto ed etico non solo all'esame, ma anche nel lavoro. CI si riferisce al capitolo sei del libro "modern data science with R".

Un'altra cosa che ha a che fare con l'etica e la riproducibilità delle analisi. Quando si fa un'analisi dei dati si registra ogni passo anche i fallimenti. In questo il formato markdown è perfetto. È molto importante documentare l'analisi tenendo traccia, questo perchè altri potrebbero voler riutilizzare l'analisi. Fondamentalmente si mette a disposizione il codice e il DSet. Per esempio Excel non tiene traccia del percorso e quindi sarà difficile condividere il percorso fatto.

Alle volte non pubblicare qualcosa è poco etico. Questo ha a che fare con il "pubblication bias", cioè fenomeno per il quale vengon riportati solo i successi e non i fallimenti. L'effetto è che apparentemente si hanno solo risultati positivi, anche se sono solo il 5% delle prove fatte. Quindi potremmo avere un'idea sbagliata perché il 95 % degli esperimenti sul medesimo farmaco ad esempio sono falliti.

Quindi la prima regola è la correttezza. Un'altra cosa è collaborare, quindi mettere apertamente i dati a disposizione e magari mettere anche il codice a disposizione. Un modo per farlo e Git e Github. Questo è lo strumento per pubblicare il progetto. Quello che si consiglia è di guardare il corso su DataCamp che riassume tutta la prossima parte.

Shiny è un modo per rendere interattiva le presentazioni. In R è possibile gestire l'interatività con Shuny e HTML widget. Questi ultimi sono dei componenti che possiamo utilizzare nel vostro markdown e che rendono le nostre visualizzazioni iterative. Queste componenti vengono elaborati a livello di client. Vengono convertiti in un codice java script che poi funziona con un qualsiasi browser moderno. Per vedere esempi di vada sul link http://www.htmlwidgets.org/showcase_leaflet.html

Questo metodo ha uno svantaggio, perché si rendono iterative solo le visualizzazioni che hanno un grafico. Se vogliamo creare noi una nostra visualizzazione dobbiamo usare Shiny.

In questo caso costruiamo noi l'interazione, quindi sia il widget che i componenti reattivi. La computazione avviene sul server. Per vedere degli esempi si vada qui: http://shiny.rstudio.com/gallery/

Qua non c'è nulla di java script, ma la computazione avviene sul server. Lo svantaggio quindi è quello di dover aver un server. Noi possiamo far girare con R Studio, esempio:

```
library(shiny)

# Define UI for application that draws a histogram ui = fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins sidebarLayout(
        sidebarPanel(
        # widget slider with name bins
        sliderInput("bins",
                    "Number of bins:", min
                    = 1,
                    max = 50,
                    value = 30)
    ),

    # Reactive output with name distPlot mainPanel(
        plotOutput("distPlot")
    )
  )
)

# Define server logic required to draw a histogram server <-
function(input, output) {

  # render the reactive output with name distPlot output$distPlot <-
  renderPlot({
      # generate bins based on input widget named bins from the UI
      x    <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)
      # draw the histogram with the specified number of bins hist(x, breaks = bins, col =
      'darkgray', border = 'white')
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

L'ultima componente che si possono usare sono i "dashboard" che sono dei cruscotti. Per degli esempi: https://rmarkdown.rstudio.com/flexdashboard/examples.html

Quindi l'interattività si può usare in questi due modi. Infine se si vuole uscire dagli standard si consigliano tre autori che son degli infografici che stanno cambiando le cose: Edward Tafti, M. Stefaner, Posavec e Giorgia Lupi.

fine