Data management for Big Data 2017/2018

Indice

1 Database relazionali 2

1.1 Modello entità relazioni 2

Time to practice (nycflights13) 6

1.2 Modello relazionale (RM) 7

Traduzione da modello ER a quello relazionale RM [ER → RM] 11

Entità 11

Relazioni 12

Per casa 14

1.3 Progettazione concettuale e logica (esercizi) 14

- Università 14
- Automobili 15
- Paesi 16
- Voli di New York 16

1.4 SQL 17

1.5 Impariamo SQL 17

Create a database in SQLite 20

Violate vincoli di integrità 22

Query 23

1.8 make SQL from R {Rmd} 37

1.9 Algebra Relazionale e Calcoli 39

Modello relazionale 39

Algebra relazionale 39

UNIONE 40

INTERSEZIONE 41

PROIEZIONE 42

JOIN 44

RINOMINA 45

Calcolo dei domini relazionali 48

SET OPERATORS 49

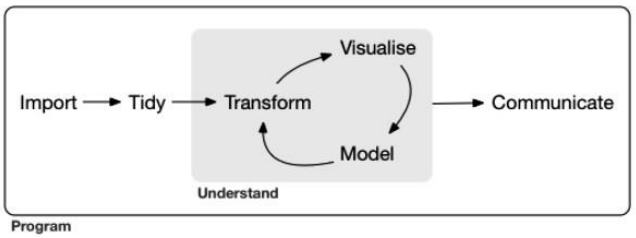
1.10 Invited speaker: Nicola Vitacolonna on data normalization {pdf} 58

5/3/18

Libro: *R for Data Science*. Hadley Wickham and Garrett Grolemund (80% del corso)

Imparerò ad organizzare, trasformare, analizzare e visualizzare piccoli e grandi dati, nonché come comunicare efficacemente i risultati del flusso di lavoro (ci tiene molto il prof a quest'ultima parte). Questo percorso di apprendimento inizia da Edgar Frank Codd e passando per Hadley Wickham raggiunge Giorgia Lupi.

Flusso tipico di data science: 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.



Molti dati non sono in forma tabellare, ma sono dati che possono essere rappresentati tramite reti o grafi. Poi si tratterà di gerarchie (alberi) con linguaggi XLM e XPATH. Vedremo anche in questo caso le difficoltà nel tabellare i dati nell'ambito *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.

Qui c'è il libro di testo: http://r4ds.had.co.nz/introduction.html (solo primo e il terzo della lista)
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* (solo due capitoli).

1 Database relazionali

Analizzeremo i database relazionali che sono i più diffusi. Esistono anche DB non relazionali. Questi però sono i più diffusi e quelli più collegati con la 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 si traducano nel linguaggio della DS. L'argomento DB è molto più ampio di quanto vedremo noi. Noi vedremo sostanzialmente 4 cose:

- Modello concettuale
- Modello logico
- modello fisico e anche per interrogare una base di dati
- Brevissimo confronto fra DataBase e Data Science

Useremo **SQLite**, un **database managment system (**DBMS), che gestisce una buona quantità di dati. Vedremo come usare pacchetti R per usare i database direttamente dall'ambiente R.

1.1 Modello entità relazioni

6/3/18

Puntualizziamo i modelli "concettuale", "logico" e "fisico":

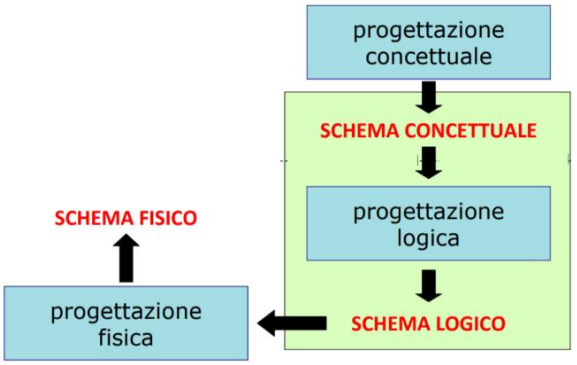
- Il livello "**concettuale"** rappresenta, in modo non necessariamente dettagliato, concetti e relazioni tra concetti. È completamente indipendente dalla tecnologia DBMS (Data Base Management System) che gestirà le strutture dati.
- Il livello "**logico**" produce un modello dettagliato, guidato essenzialmente dalle relazioni di significato tra i dati. È il livello di definizione delle tabelle in un DBMS relazionale, con indicazione di colonne e data type, chiavi primarie ed alternative, regole di integrità. A differenza del modello concettuale, può essere

un modello ottimizzato. Può, ad esempio, comprendere ridondanze introdotte per migliorare le tempistiche di accesso in consultazione. Il modello logico è l'input per la progettazione fisica dei DB, ed è opportuno che raggiunga il massimo livello di precisione possibile. Il modello logico deve, cioè, contenere tutte le informazioni necessarie che non sono legate a competenze di natura tecnica, ma che derivano invece dalla conoscenza approfondita dell'ambito applicativo, e quindi del significato dei dati.

- Il livello "**fisico**" è quello in cui si definiscono caratteristiche utili per l'ottimizzazione delle prestazioni e della memoria (es. indici, percentuali di spazio disponibile per inserimento di nuove righe) o per la gestione del DBMS (es. organizzazione in data base e table space).

L'ordine con cui si costruiscono i modelli è: (CLiF x ricordarlo)

1) modello concettuale
2) modello logico
3) modello fisico



Il modello **Entity-Relationship** (**ER**) è un **modello concettuale** (1)) dei dati. Come tale descrive ad alto livello la realtà modellata indipendentemente da come i dati saranno rappresentati logicamente e fisicamente. Il modello ER definisce uno **schema concettuale** di dati.

Una differenza di concetto molto importante è quella fra **schema** ed **istanza**: lo schema corrisponde alla nozione di classe, mentre un'istanza dello schema è un oggetto della classe. È come la differenza fra classe ed oggetto, oppure fra progetto e realizzazione. Vediamo questi concetti. Uno **schema** descrive la struttura o forma dei dati, ma non dice nulla sui dati (su istanze e occorrenze dello schema). Ad esempio, lo schema ER può dire che l'entità piano ha gli attributi produttore del piano e modello di pianoforte, ma non dice quali siano i valori effettivi di questi attributi. In sintesi: uno schema descrive la forma dei dati, ma non descrive i dati. Le **istanze** sono i dati inseriti nello schema. Quando parleremo di **modello concettuale** ci riferiremo allo **schema concettuale**.

Per costruire un modello concettuale, quindi anche un modello ER, dobbiamo decidere due cose:

- **Entità**. rappresentano un complesso e rilevante concetto con esistenza indipendente. Una istanza di una entità è un oggetto della classe rappresentata. Ad esempio, le possibili entità in un database di volo sono: voli, aeroplani, aeroporti, compagnie aeree, condizioni meteo.
- **Relazione**. rappresenta un legame logico, significativo per la realtà modellata, tra due entità. Un'istanza di una relazione è una coppia di istanze di 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 aeroplano (o viceversa).

Per progettista: 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 più sotto.

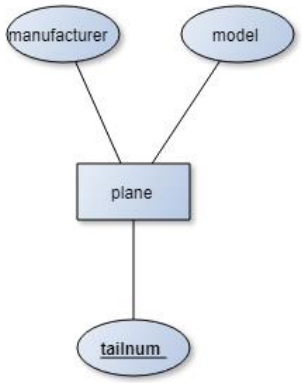
Sia le entità che le relazioni possono avere proprietà atomiche chiamate **attributi**. Gli attributi possono essere **obbligatori** (deve essere fornito un valore) o **opzionali** (il valore potrebbe non esistere o essere sconosciuto).

Un attributo dovrebbe avere un nome univoco per essere riconosciuto (ma posso creare una base di dati con attributi aventi stesso nome purché si riferiscano ad entità distinte).

L'attributo che è sottolineato è una **chiave**, cioè un "codice" cioè un insieme di attributi che identifica univocamente l'istanza dell'identità. Un attributo-chiave o solo chiave deve avere due requisiti:

- deve essere **obbligatoria**, cioè non posso lasciarla vuota
- deve essere **minimale**, cioè 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 all'identificazione un attributo inutile.



rettangoli = entità

rombo = relazione

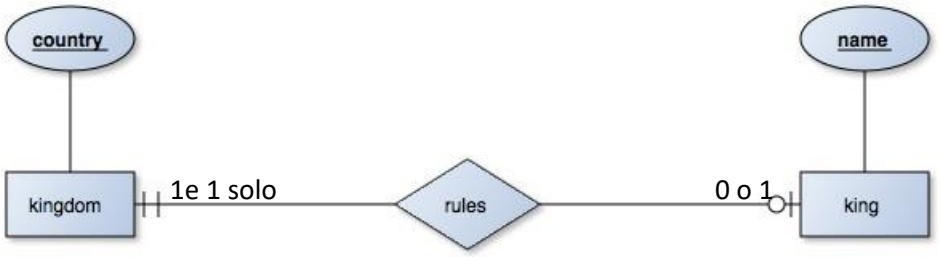
ellissi = attributi o proprietà; attributi sottolineati = chiave primaria

Qui l'entità aeroplano ha gli attributi produttore, modello ed è identificato dall'attributo chiave tailnum (= numero di serie dell'aeroplano)

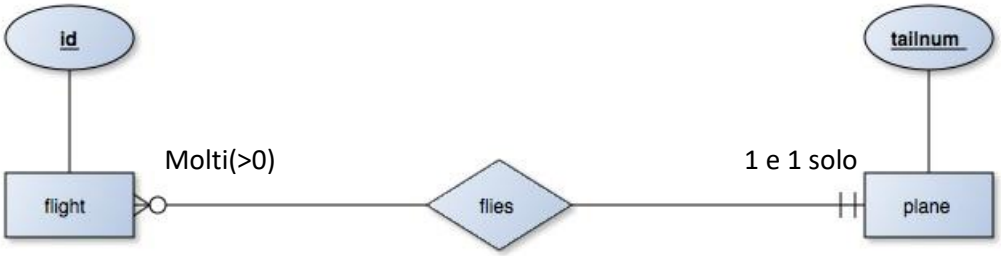
Il secondo concetto del modello ER sono le **relazioni**. Le relazioni rappresentano un legame logico significativo ed importante per la realtà che andiamo a modellare tra due entità. Consideriamo due entità perché vediamo solo relazioni binarie. Questa scelta è dovuta dal fatto che sono le più frequenti, pur esistendo relazioni con più di due entità.
Per esempio, ci può essere una relazione tra i voli e gli aeroplani. Quindi dobbiamo legare queste due entità. Le relazioni servono a questo. Questo è un ambiente molto potente, perché in termini di entità e relazioni si possono tradurre moltissime situazioni.

La relazione è indicata con il rombo. Ci sono 3 tipi di relazioni:

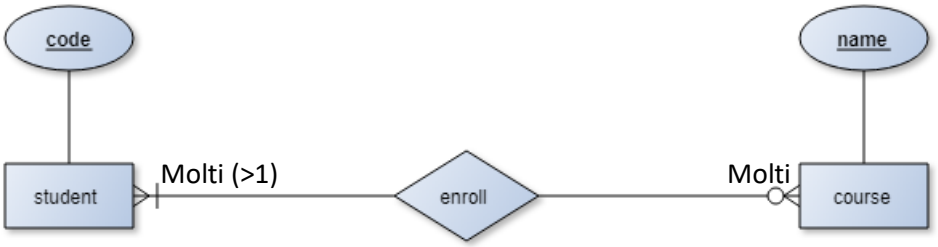
- **Relazione uno a uno**: è sostanzialmente una funzione. Come se per esempio abbiamo un regno ed un re, la relazione è "governare", in questo caso ogni re governa un regno e ogni regno è governato da un re, quindi non posso avere regni con più re o un re con più regni. Se ci pensiamo è una funzione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



- **Relazione uno a molti**: un'istanza di un'entità può essere associata a molte istanze, ma non viceversa. Un'istanza volo "vola" (viaggia in aeroplano) attraverso una ed al più una sola istanza aeroplano. Ma un'istanza aeroplano può volare con 0, 1 o + voli.

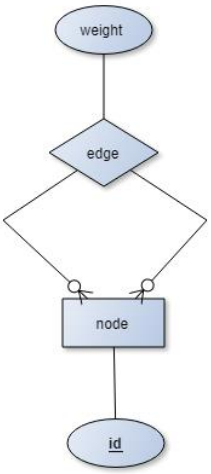


- **Relazione molti-a-molti**: un'istanza di un'entità può essere associata a più istanze e viceversa. L'esempio classico è la relazione tra studenti e corso. L'istanza studente è iscritta a 0 corsi (se piano di studi è da fare) o più corsi, mentre ogni corso può essere frequentato da 1 o più studenti.

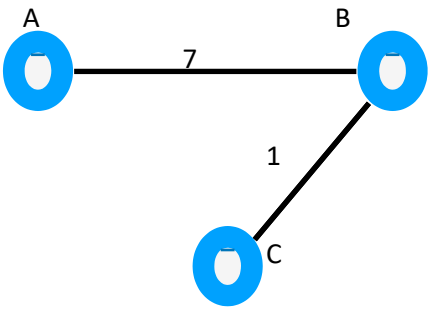


Per progettisti: questi vincoli li poniamo noi, nella fase preliminare chiamata **fase dei requisiti**, decido tutte queste caratteristiche. Sembrano concetti banali, ma il tutto è molto potente e serio, se si comprendono bene questi tre concetti possiamo modellare molte situazioni, e possiamo capire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

Nulla vieta di avere **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.



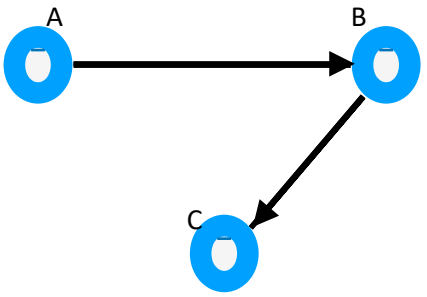
Un esempio di **grafo** è la rete sociale, come Facebook. Su Facebook abbiamo dei nodi che sono le persone. Per esempio, prendiamo la rete Facebook con tre persone. Gli archi sono i collegamenti (le amicizie nell'esempio). Risulta che A è amico di B, B è amico di C e C ed A non sono amici.



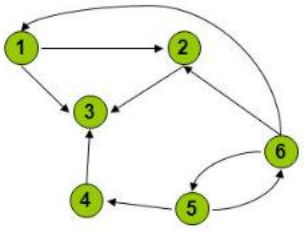
pallini = nodi = entità, persone
archi = linee fra nodi = relazione di amicizia

Nel grafico sopra di nodi e archi 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 alla coppia di nodi: come nel grafico A e B si conoscono da 7 anni e B e C da solo 1. Parliamo allora di **grafi pesati**.

Se fossimo in Twitter avremmo degli archi orientati in cui A segue B, e B segue C. Parliamo allora di **grafi orientati**.



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.



6/3/18

Time to practice (nycflights13)

Il set di dati nycflights13 contiene informazioni su tutti i voli che sono partiti da New York City (ad esempio EWR, JFK e LGA) nel 2013 e relativi metadati (aeroplani, aeroporti, compagnie aeree e condizioni meteo). Include le seguenti entità:

- **voli**: voli in partenza da New York nel 2013. Gli attributi includono
  - Id volo, anno, mese, giorno, ora, numero volo, origine, destinazione, orari di partenza e arrivo effettivi, orari di partenza e arrivo programmati, ritardi di partenza e di arrivo, distanza. La chiave primaria è id di volo.
- **aeroplani**: informazioni di costruzione su ciascun aereo.
  - Gli attributi sono: numero di coda, anno di fabbricazione, tipo di aereo, costruttore, modello, numero di motori e sedili, velocità di crociera media, tipo di motore. La chiave primaria è il tailnum.
- **aeroporti**: nomi et località dell'aeroporto.
  - Gli attributi sono: codice dell'aeroporto, nome dell'aeroporto, ubicazione dell'aeroporto, fuso orario, altitudine. La chiave primaria è il codice di aeroporto.
- **compagnie aeree**: traduzione tra codici e nomi di due lettere portanti. La chiave primaria è il codice del vettore.
- **meteo**: dati meteorologici orari per ciascun aeroporto,
  - Gli attributi sono: aeroporto, anno, mese, ora, temperatura, punto di rugiada, umidità, direzione del vento, velocità e velocità di raffica, precipitazioni, pressione, visibilità. La chiave primaria è la combinazione di aeroporto, anno, mese giorno, ora.

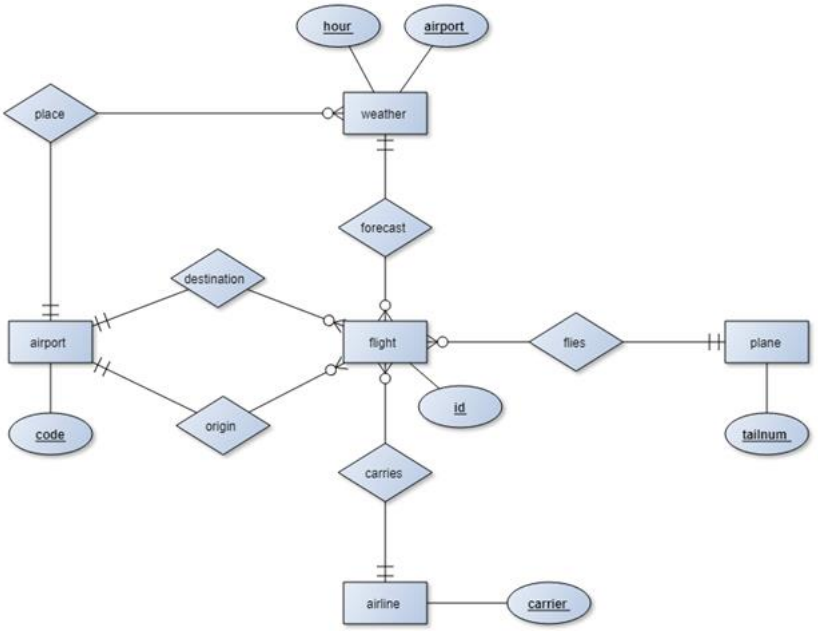
Le entità sono associate alle seguenti relazioni:

- una relazione che associa un volo con l'aereo con cui volava
- una relazione che associa un volo alla compagnia aerea con cui ha volato
- una relazione che associa un volo con l'aeroporto di origine
- una relazione che associa un volo con l'aeroporto di destinazione
- una relazione che associa un volo alle condizioni meteorologiche dell'origine e all'orario di partenza
- una relazione che associa una condizione meteorologica ad un aeroporto

Consegna: costruisci uno schema ER per il set di dati sui voli di New York. Aggiungi un sottoinsieme degli attributi descritti che include gli attributi chiave. È possibile utilizzare carta e matita o editor grafico Yed per disegnare il diagramma. https://www.yworks.com/products/yed

Vediamo qua sotto la soluzione a questo problema.

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



Commenti:

- un aeroplano può volare per più voli, ma un volo per volare deve avere un solo aeroplano
- una compagnia aerea può far volare più voli, ma un volo vola con una sola compagnia aerea
- un volo ha bisogno di un unico aeroporto par partire ed un unico aeroporto per atterrare, ma da un aeroporto partono e arrivano 0 o più aerei
- meteo: le sue chiavi sono il luogo e ora (per ogni ora e per ogni aeroporto è dato il meteo). Il volo che parte è associato ad un tempo (da Venezia alle 19.40 con rispettivo meteo), mentre le condizioni atmosferiche posso riferirsi a più voli (un solo meteo per più voli che partono da un luogo).
- c'è un solo meteo per un aeroporto, mentre un aeroporto può essere associato a diversi meteo

5/3/18

1.2 Modello relazionale (RM)

Il modello relazionale è un modello logico (2)). Un modello ER diventerà un modello logico, quindi relazionale per noi, che poi diventerà modello fisico. Il modello logico è indipendente dal modello fisico, che è la rappresentazione fisica dei dati.

Il modello logico che noi useremo è chiamato **modello relazionale (RM)**. 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 componenti:

- **Strutture per organizzare i dati** (es dati, tabelle o relazioni, che poi sono sinonimi)
- **Vincoli di integrità** che consentono di mantenere consistente la base di dati

Storicamente: Il modello relazionale fu proposto nel 1970 da Edgar F. Codd in un articolo. Code si occupava di un modello di dati che fosse indipendente dall'implementazione, cercava un modello che pur cambiando implementazione non cambiasse modello logico. Oggi è ovvio che non cambia il modello logico al cambiare dell'implementazione. Il suo modello è basato su relazioni n-arie chiamate **relation** ossia attributi (*relation* nell'articolo si riferisce agli attributi; è diverso da *relationship*). Inoltre Code propose un *unserval laguage data* cioè un modello universale: era il germe dell'idea la cui implementazione portò al linguaggio SQL.

Il modello è basato sul concetto di **relazione** (ocio! da non confondere con la *relazione concettuale* del modello ER), la cui rappresentazione è una tabella (o un *dataframe* in gergo R).
Nota terminologica: "*relation"* → se parliamo di modello logico, mentre "*relationship"* → se parliamo di modello concettuale (i rombi dello schema). **Relazione è sinonimo di tabella**, infatti quando costruisco una tabella metto in relazione le variabili. Possiamo vedere le tabelle come dei predicati in logica o delle relazioni in algebra. Il concetto di relazione qui è formale e deriva dalla teoria degli insiemi. La popolarità del concetto di relazione o tabella è dovuta sia alla semplicità nell'intuire come funziona, sia alla profondità del concetto. Il successo del modello relazionale sta proprio nella congiunzione di un concetto formale, la relazione (che ha permesso lo sviluppo di una teoria dei database relazionali con risultati di impatto pratico) con il concetto intuitivo della tabella (che ha reso possibile il modello relazionale anche per gli utenti finali senza alcuna nozione matematica). L'approccio utilizzato di seguito sarà intuitivo.

Ci sono alcune caratteristiche di una relazione da presentare:

- Le **colonne** sono degli **attribuiti** con un nome univoco per ogni colonna
- Le **righe** (dette ***tuple*)** sono le **unità** 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 interpretate come **insiemi**.

Assumendo le colonne come delle variabili, ad ogni colonna è associato un supporto cioè l'insieme delle possibili realizzazioni di una variabile. Possiamo avere domini diversi per ogni tipo di dato. Il dominio è *atomico* cioè indivisibile.

Nb. Non si possono inserire righe duplicate, questo vuol dire che ci deve essere sempre una **chiave.** 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*, che si usa come chiave.

Gli attributi possono avere valori nulli, cioè non presenti (dato mancante). I motivi possibili sono 2: il dato non è reperibile, cioè non esiste, o esiste ma non ce l'ho (es. non ho la mail del candidato). Si ricordi che "*NULL"* è costante nulla nei DB mentre "*NA"* è la costante nulla in R.

Relazione (R) è una **tabella**
Colonna è una **variabile**
Riga (o **tupla**) è un'istanza della entità oppure istanza della relazione. Corrisponde ad una **unità**.

- Esempio:

attributo

ennupla o tupla

Casa	Fuori	RetiCasa	RetiFuori
Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	1	2
Roma	Milan	0	1

```
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

In una relazione distinguiamo lo schema della relazione dalla istanza dello schema (o istanza della relazione):

- Lo **schema della relazione R(X)** è composto dal 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** (dello schema) invece è un insieme di tuple della relazione. L'istanza parla dei dati, perché essa stessa è i dati, invece lo schema non tratta dei dati.

Tipicamente vedremo che ci sono più tabelle che vengono combinate con una operazione join, esse parlano tramite il meccanismo delle chiavi e delle chiavi esterne. Per rispondere alle domande a cui si è interessati quello che si fa non è mettere tutto in unica tabella, ma in più tabelle come nell'esercizio precedente, una per ogni entità concettuale, e poi si cercherà un modo per collegarle. Il modo consiste nel collegarle attraverso una chiave esterna detta foreign key. Collettivamente, tabelle multiple di dati in relazione sono chiamati ***database relazionale***. Sia le relazioni che le entità verranno trasformate in tabelle, quindi avremo solo tabelle.

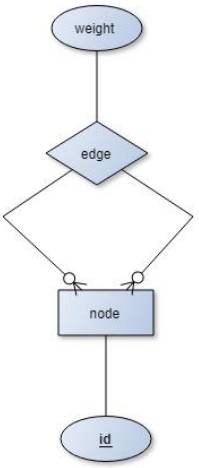
Perciò definiamo:

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

Nel modello relazionale, tutta l'informazione è rappresentata tramite una relazione che corrisponde a una tabella con gli attributi.

Non c'è più la distinzione fra tabella e relazione perché abbiamo solo tabelle. 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:



Possiamo usare solo le tabelle. L'idea è creare due tabelle:
- Una per i l'entità *nodo*
- Una per l'entità *arco* → ha due puntatori, sue chiavi, che puntano ad id di node.

> node(id, name)
> edge(from, to, weight)

La chiave "from, to" è una chiave della tabella edge.

Abbiamo tradotto una tabella molti a molti in una tabella che contiene le coppie delle chiavi dei nodi che partecipano alla relazione *chiave del nodo predecessore* e *chiave del nodo successore*. From è una chiave esterna perché è propria di node però vive in edge. Questo schema concettuale possiamo trasformarlo in schema logico

Un esempio di istanza di queste due tabelle:

```
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
```

Un **grafo** è una rete indiretta. Un grafo è un insieme di nodi e di archi, che sono coppie di nodi. A ciascun nodo è un attribuito peso che identifica la forza della relazione. Questo schema concettuale possiamo trasformarlo in schema logico. Questo sottostante è un esempio di un nodo contenente un ciclo di lunghezza 6. Un esempio di istanza della tabella nodi e di istanza della tabella edge è:

```
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. Si noti che i valori delle chiavi esterne devono essere valori che ritrovo nella tabella originaria: colonne from e to nelle tabelle edge si riferiscono a valori id validi nella tabella dei nodi.

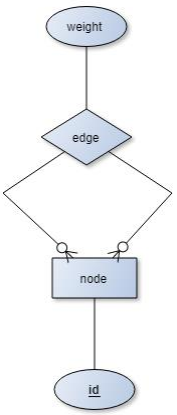
12/03/18

Il modello relazionale è formato da relazioni e **vincoli di integrità**, che servono a mantenere la consistenza del database. Sono valide solo le istanze del database che soddisfano tutti i vincoli di integrità. I vincoli di integrità sono:

- **Vincoli di dominio**: specificano il dominio di un attributo (un dominio stringa non può avere integer)
- **Vincoli chiave**: specificano che una *chiave primaria* deve avere valori univoci e non nulli.
- **Vincoli di chiave esterna**: ogni valore non nullo della chiave esterna deve corrispondere ad un valore esistente allo schema a cui è riferita. Specificano che i valori non nulli della *chiave esterna* devono corrispondere ai valori della chiave a cui la chiave esterna si riferisce (chiave di riferimento).

I **vincoli** si scrivono con delle frecce in questo modo:

> edge(from) –> node(id)
> edge(to) –> node(id)



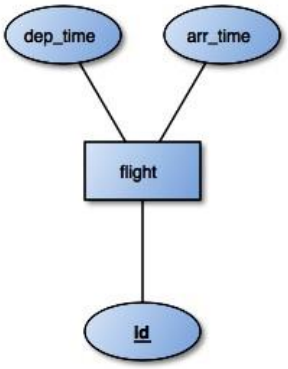
nome tabella (chiave esterna) —> nome della freccia riferita (chiave riferita da chiave esterna)

Traduzione da modello ER a quello relazionale RM [ER → RM]

Esiste ora una traduzione da ER (primo dei 3 modelli) a RM (secondo dei 3 modelli). È un algoritmo che traduce uno schema di entità relazioni in uno schema logico relazionale. Vedremo un insieme di regole per questa traduzione. Funziona mappando entità e relazioni dello schema ER concettuale in relazioni e vincoli di integrità dello schema RM logico. Le regole sono molto semplici, vediamole una per volta.

Entità

Una entità è una tabella con medesimo\i attributo\i dell'entità di partenza.
La si traduce con una tabella con lo stesso nome dell'entità e con gli stessi attributi. L'attributo chiave sarà sottolineato.



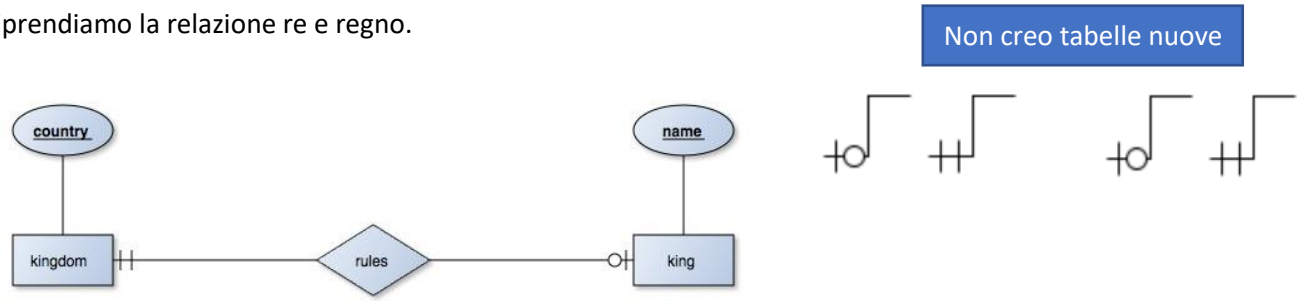
> **flight(id, dep_time, arr_time)**

Relazioni

Come tradurre le relazioni? È delicato. Le relazioni del modello concettuale le trasformerò in tabelle del modello logico. Nella maggior parte dei casi non dovrò creare una tabella per la relazione perché la relazione sarà inserita in una delle due tabelle delle entità, ma altre volte dovrò farlo perché altrimenti violerei il vincolo di chiave primaria.

RELAZIONI UNO A UNO

Riprendiamo la relazione re e regno.



Soluzione 1:
nella prima riga butto la relazione nella tabella king.

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

Qui è associato ad ogni re al massimo un unico paese su cui regna.

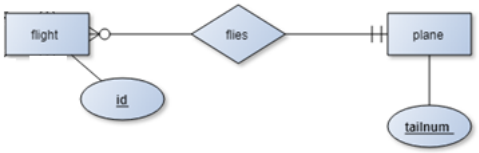
Soluzione 2:
Oppure creo un altro attributo che chiamo *country* che è una chiave esterna che si rif alla country di king. In questo secondo caso butto la relazione nella tabella kingdom. Qui aggiungo un attributo alla tabella kingdom che è una chiave esterna che si riferisce al nome del re

> king(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 e associare valori nulli in un DB non è bene perché è una mancanza di informazioni.
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.

RELAZIONI UNO A MOLTI

Prendiamo la relazione tra flight e planes.



> flight(id, tailnum)
> plane(tailnum)
> flight(tailnum) –> plane(tailnum)

Non creo tabelle nuove

Regola: la relazione va inserita dentro all'entità che partecipa con vincolo di integrità massimo 1.

Qui tailnum è chiave esterna. La traduzione giusta è quella che ingloba la relazione flies all'interno di flight
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.

NB (da cancellare una volta capito)
Perché non ho messo flight nella tabella plane?
Perché perderei il vincolo chiave primaria, perché se traducessi :

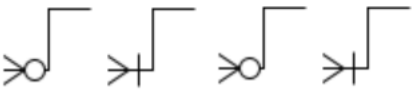
plane(tailnum, flight)
flight(id)

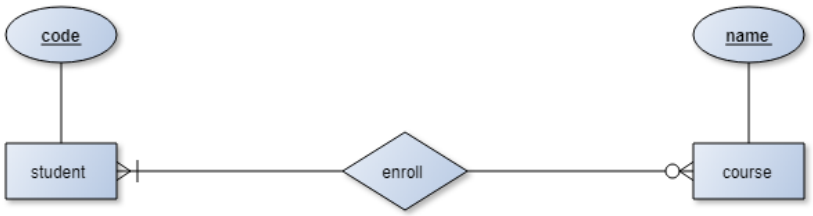
→ ad ogni aereo posso associare più voli! Prendo un aeroplano che conduce 2 voli, allora quelle 2 righe avrebbero attributo tailnum duplicato, che quindi non sarebbe più una chiave in quanto non identifica più le righe della tabella.

RELAZIONI MOLTI A MOLTI

Prendiamo la relazione tra studente e corso. In questo caso non posso aggiungere la relazione in nessuna delle 2 tabelle esistenti, perché violerei il vincolo di chiave. Quindi devo creare una nuova tabella, la cui chiave è la combinazione delle chiavi delle due entità.

Creo una nuova tabella

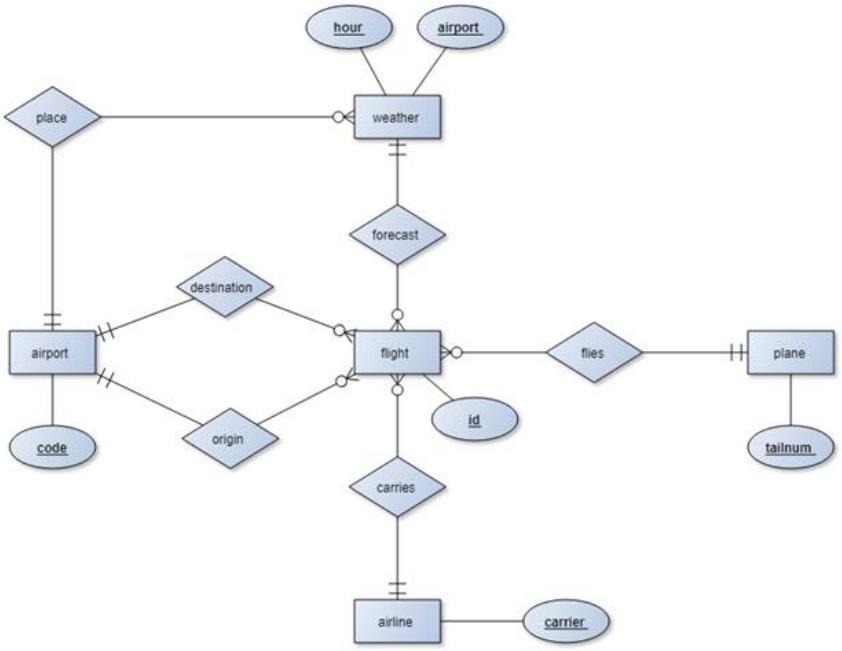




> student(code)
> course(name)
> enroll(student, course)
> enroll(student) –> student(code)
> enroll(course) –> course(name)

Regola: la chiave della nuova tabella è la combinazione delle chiavi delle entità

Tornando all'esempio dell'aereo:



flight(id, month, day, hour, origin, destination, tailnum, carrier)
plane(tailnum)
airline(carrier)
airport(code)
weather(time_hour, origin)
flight(tailnum) –> plane(tailnum)
flight(carrier) –> airline(carrier)
flight(origin) –> airport(code)
flight(destination) –> airport(code)
flight(time_hour, origin) –> weather(time_hour, origin)
weather(origin) –> airport(code)

Sintesi regole:

1. **relazione 1 a 1: traduco la reazione buttandola dentro l'entità**
2. **relazione 1 a molti: butto dentro la relazione l'entità che partecipa con vincolo di integrità max 1 (ocio)**
3. **relazione molti a molti: creo nuova tabella**
4. regola per l'entità:

Per casa
http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/make/ER/ER.html

Per ogni universo del discorso (Università, Automobili Paesi):

I. Scrivi un modello concettuale (ER) per l'universo descritto
II. Tradurre il modello ER in un modello relazionale che includa i vincoli di integrità
III. Scrivi file CSV coerenti per ogni tabella
IV. Apri RStudio e leggi i file CSV creati in frame di dati con la funzione read_csv ()
V. Visualizza i frame di dati caricati con la funzione Visualizza ()

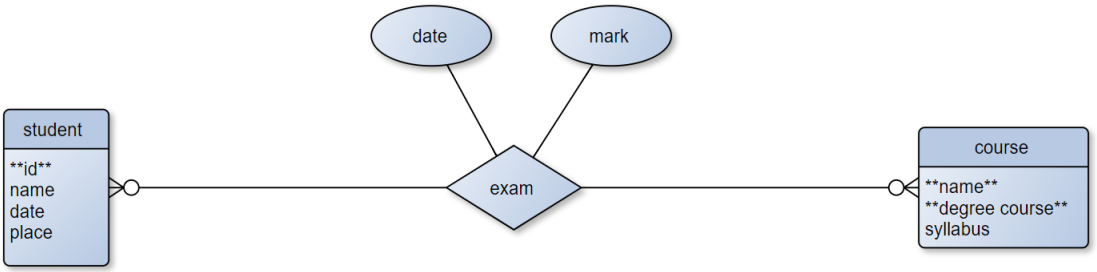
1.3 Progettazione concettuale e logica (esercizi)

- ### Università

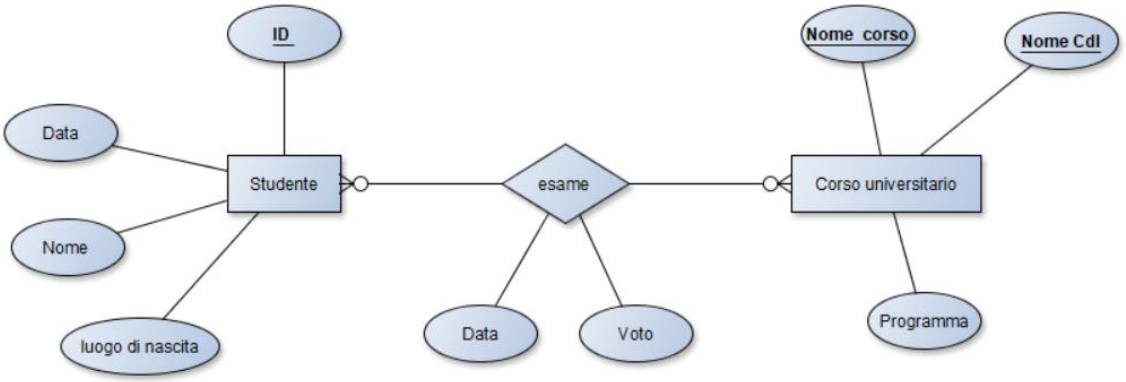
  Uno **studente** è identificato da un ID e descritto per nome, data e luogo di nascita.

Un **corso di insegnamento** è identificato dal nome e dal nome del corso di laurea a cui appartiene ed è ulteriormente descritto da un programma.
Un **esame** associa gli studenti ai corsi e registra la data e il punteggio dell'esame.

Soluzione



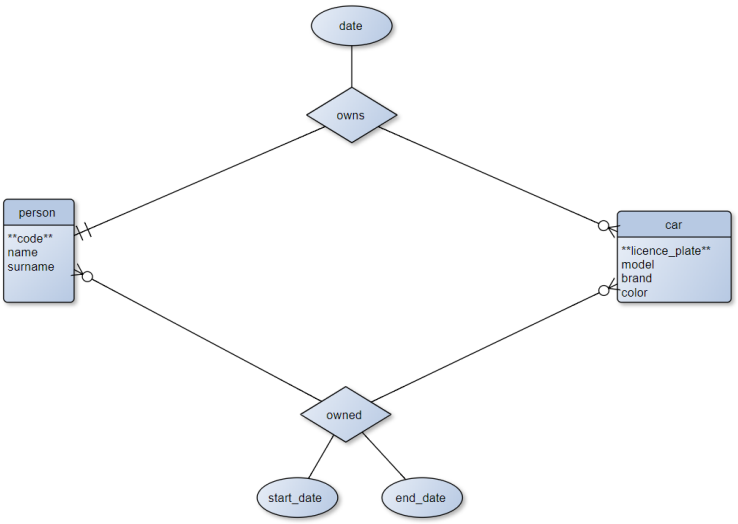
> student(<u>id</u>, name, date, place)
> course(<u>name</u>, <u>degree_course</u>, syllabus)
> exam(<u>student</u>, <u>course</u>, <u>degree_course</u>, date, mark)
> exam(student) –> student(id)
> exam(course, degree_course) –> course(mame, degree_course)



- ## Automobili

Un' **auto** è descritta da una targa, un modello, un marchio e un colore. Ogni auto ha un unico proprietario che ha acquistato l'auto in una determinata data. Il proprietario è una **persona** descritta da codice fiscale, nome e cognome e può possedere più automobili.
Si desidera tenere traccia dei proprietari precedenti di una macchina e dell'intervallo di tempo (date di inizio e fine) in cui possedevano la macchina.

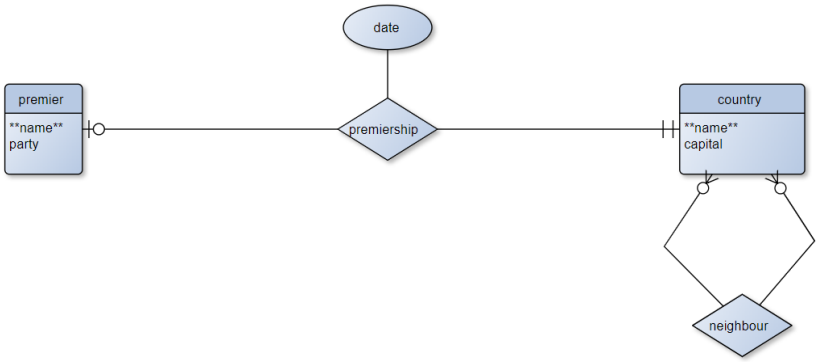
Soluzione



> person(tax_code, name, surname)
> car(license_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)

- ## Paesi

  Un **paese** è identificato da un nome e ha una città capitale. Un paese ha un premier con un nome, una data di elezione e un partito politico. Un Paese potrebbe confinare con altri Paesi.



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

- ## Voli di New York

Traduci lo schema ER qui sotto per il set di dati di volo di New York in un modello relazionale. Per semplicità, effettua la traduzione solo per gli attributi chiave. (20 minuti a disposizione)

Soluzioni

> flight(id, month, day, hour, origin, destination, tailnum, carrier)
> plane(tailnum)
> airline(carrier)
> airport(code)
> weather(time_hour, origin)
> flight(tailnum) –> plane(tailnum)
> flight(carrier) –> airline(carrier)
> flight(origin) –> airport(code)
> flight(destination) –> airport(code)
> flight(time_hour, origin) –> weather(time_hour, origin)
> weather(origin) –> airport(code)

Registrazione 245

1.4 SQL

Si scarica da qui https://www.sqlite.org/index.html

1.5 Impariamo SQL

SQLite è un **DBMS** (Data Base Management System), molto usato in data science. È un linguaggio di interrogazione e manipolazione 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** (Structured Query Language) 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 le ***query*** (interrogare, domandare) ossia il recupero dei dati. Avuti i dati vogliamo un linguaggio che permette agevolmente di lavorarci su. SQL è stato uno dei primi linguaggi commerciali per il modello relazionale di Edgar F. Codd.

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 spesso che i dati non stanno in memoria e serve SQL per estrarli per poi essere elaborati in R o in Python.

SQL è un **linguaggio dichiarativo**, mentre C e Java sono imperativi (fai questo ciclo, se .. allora.. fai questo). Programmatore dichiara solo quello che gli interessa il resto lo fa SQL. Si dichiara solo il proprio intento, si specifica cosa vogliamo recuperare ma non diciamo 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.

SQLite ha le seguenti caratteristiche

- Il database con anche 1 000 000 di tabelle è salavato in un unico file
- Il code è molto piccolo quanto a dimensione
- Ha interfaccia di programmazione molto facile
- È veloce
- Scritto in un solo file C
- Utilizzato in tutte le piattaforme

- 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.

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 ***commit*** (utile!) che rende effettive le richieste oppure possiamo tornare indietro, con l'istruzione ***roll back*** (disfa tutto il lavoro fatto su DB), in quest'ultimo caso il DB rimane consistente.
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* (Pacchetto R)

Installare e caricare i pacchetti.

Scarica SQLite e segui il video https://www.youtube.com/watch?v=zOJWL3oXDO8
Da cartella mydir, "cmd" e "sqlite3 test.db"

Andiamo al database su R Studio.
Apri su R un file SQL da sito prof e mettila nella directory.
Library(RSQLite) → per leggere bene fiel SQL
Il file SQL lo apro su R ma poi i comandi li esegue la shell.

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.

Importante DEVO ASSICURARMI CHE I VINCOLI DI CHIAVI PRIMARIA SIANO SODDISFATI, SENNO' DEVO AGGIUNGERE DELLE CHIAVI SURROGATE.
Ora voglio importare qst database di R in una base di dati. Come? Salvare tutto con un file csv.

Create a database in SQLite

A titolo di esempio riporto istruzioni di creazione tabella

```
CREATE TABLE pippo (
 faa VARCHAR(3),
 name VARCHAR(60),
 lat REAL,
 lon REAL,
 alt INTEGER,
 tz INTEGER,
 dst VARCHAR(1),
 tzone VARCHAR(60),
 primary key (faa)
);
```

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:

Si apre un command line
.help → aiuto in linea: da i comandi possibili
NB Devo posizionarmi nella cartella dove stiamo lavorando con R, che sarà dentro una sopra-cartella col nome del progetto.

Creare tabelle con vincoli di chiave primaria e straniera

Qui usiamo SQL come linguaggio di definizione dei dati. I comandi per creare lo schema e i vincoli di integrità sono contenuti nel file *create.sql*

```
.read create.sql
```

Per sapere il nome del database scrivere

```
.databases
```

Mentre per sapere le tabelle create

```
.tables
```

Pra dobbiamo creare lo schema NON le istanze:

Scaricare dal sito il file *create.sql* mettetelo nella cartella di lavoro e apritelo da rstudio
http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/learn/database/SQL.html

13.3.18

Data definition and management

Caricare il dataset su memoria centrale solo se il pc riesce a processarlo, sennò bisogna salvare su hard disk e richiamarlo

Lista dei 10 comandamenti per SQL.

1. scarica e compila SQLite (agigungeer DBMS)
2. aggiungi la chiave surrogata ai voli tabella del set di dati

3. esportare i frame di dati in file CSV
4. creare un database in SQLite
5. creare tabelle (con vincoli di integrità) chiave primari e stranieri
6. inserire i dati nelle tabelle utilizzando i file CSV esportati
7. creare indici
8. fare un backup del database
9. provare a violare i vincoli principali e esterni della chiave con l'inserimento, la cancellazione e l'aggiornamento
10. ripristinare ("restore") il database dal backup

Su prompt comandi *cd* C:\Users\Luca\Documents\R\Esercizio SQL 13.3.18

Importo file csv create in R così le legge come tabelle

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

Replace empty values (NA values) with NULL values:

```
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 = "";
update flights set tailnum = NULL where tailnum = "";
update flights set dest = NULL where dest = "";
```

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

Ci sono 4 destinazioni non presenti nella tabella aeroporto

Il dataset non soddisfa i vincoli di integrità referenziale. Cosa posso fare?

- Non posso buttare via le info per una cosa inconsistente
- Non faccio nulla, mi tengo le chiavi esterne non integre → caso visto in aula
- Posso infine eliminare solo una parte (metto NULL) → la cosa più corretta da fare
- Aggiungo il tailnum mancante nella tabella planes

Notiamo che il database non ha protestato, ciò perché non ho ancora attivato i vincoli id chiave esterna.
Regola:

- Se vincolo chiave primaria non verificato → eliminare i duplicati
- Se vincolo di chiave differita non è verificato → o non faccio nulla, o cancello intera riga (perdo informazioni), o metto a NULL l'attributo della tabella esterna che viola vincolo di integrità referenziale o infine aggiungo l'attributo mancante nella tabella.

Create indexes

Creiamo degli indici che rendono più semplice l'interrogazione.

```
create unique index airlines_index on airlines(carrier);
```

```
create unique index
```

creo indici su chiavi primarie.

E' buona norma creare indici anche sulle chaivi straniere

```
create index flights_tailnum on flights(tailnum);
create index flights_origin on flights(origin);
```

Eseguiamo il backup

```
.backup NomeDatabese_backup_AnnoMeseGiorno
```

Nella cartella dovrebbe esserci un file con questo nome. E' il nostro database.

Violate vincoli di integrità

Selezioniamo tutte le airlines

```
select * from airlines;
```

Siccome esiste una airlines con codice 9E, ne inserisco una con nome 9E

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

Sintassi

```
insert into NOME TABELLA values ("nuovo nome", "My Air");
```

dà errore → non ho righe duplicate

Proviamo ora a violare un vincolo di integrità referenziale:
Con questo comando non posso più violare vincoli di integrità secondaria

```
PRAGMA foreign_keys = ON;
```

Provo ad inserire in Flights un nuovo volo con ID 0 (che so non esistere) e carrier xy

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

Dà errore per fallimento del vincolo di chiave esterna.

```
Oss Quando scrivo values ("0", "XY") tutti gli altri campi di values hanno valore
NULL
```

Proviamo nuovamente a violare i vincoli di integrità referenziale: cancello dalla tabella airlines il carrier UA, quindi se vado a eliminarla ottengo uno stato inconsistente

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

```
delete
```

→ elimina tutte le tuple che soddisfano il vincolo "UA"

Allo stesso modo se provo ad aggiornare il campo UA con il campo XY

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

Restore from backup

Con questo comando recupero lo stato del database ad una certa data

```
.restore nycflights13_backup_20180215
```

Fine della parte di SQL per la manipolazione dei dati.

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.

Registrazione 248 (13\3\2018)

SELECT FROM

implementa una sorta di estrazione preliminare che ci interessa poi esplorare in modo approfondito. La prima clausola è ***select from***. Select significa seleziona alcuni campi (colonne) 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.

Seleziona tutti i campi di flights

```
select *
from flights
```

```
select id, flight, carrier
from flights
limit 10
```

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 semplice come una uguaglianza o più complicata. Con questa clausola where immettiamo un **predicato booleano** sulle tuple.

Esempio: aeroplani con almeno 2 motori

```
select *
from planes
where engines > 2
```

Per fare un commento su SQL si faccia il doppio trattino --

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* ***op*** *Y*, dove X e Y sono delle costanti o sono delle variabili (nomi colonne) ***op*** è un operatore. Gli operatori possibili sono:

=
<>
<, >
<=, >=
like
"_" → un solo carattere
"%"→ vale per ogni valore, anche più lungo di una lettera

Esempio: NOME like "_MA" allora trova ad esempio "AMA" (non trova MAMA)
Esempio: NOME like "%MA" allora trova "MAMA"

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. Se voglio verificare che il predicato di un attributo è nullo o non è nullo:

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 flights
Where month =12 and day= 25
Limits 50
```

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 (id, dep_delay, arr_delay)
From flights
Where dep_delay is null or arr_delay is NULL
Limit 20
```

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 (id, dep_time, arr_time)
From flights
Where (dep_time is not null and arr_time is not null)
```

Voli che hanno la variabile tailnum che inizia con N10:

```
Select *
From planes
Where tailnum like "N10%"
```

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 né valore vero né 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 (engine <= 2)
```

Questa query potrebbe non selezionare tutti gli aeroplani, perché where seleziona con le condizioni vere.

Se ci fossero dei valori nulli, confrontando valori nulli con una costante avremmo come risposta nullo, ma siccome vogliamo come risultato quelli con risultato TRUE, vengono selezionati 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 uso la parola chiave **desc**

```
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*. A partià di nome conta l'ordine del cognome.

Proviamo ora a fare i voli che hanno ritardo positivo ordinati in ordine decrescenti per ritardo:

```
Select id, dep_dealy
From flights
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
```

Dove *(arr_delay-dep_delay)* ***as*** *catchup* è una nuova variabile create per l'interrogazione. Ho definite una nuova variabile che posso utilizzare per il resto della interrogazione.

Introduzione a GROUP BY e HAVING

Ci sono degli operatori aggregati che calcolano delle statistiche su un insieme di tuple (es media, massimo, minimo), il risultato è un numero singolo. Un operatore di aggregazione è una funzione che immette un set di tuple in una tabella e genera un valore atomico. Ad esempio, se voglio calcolare la media dei voti dei maschi e delle femmine, devo usare questi operatori perché voglio due gruppi. Lo standard SQL fornisce i seguenti **operatori di aggregazione**: *count, min, max, sum, avg*. Questi operatori si possono applicare a tutta la tabella oppure a dei gruppi.

Per contare 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.

Come creare i gruppi?

GROUP BY

Uso la clausola **group by** e farla seguire dal campo (o più campi). Un gruppo viene creato per tutte quelle righe che hanno lo stesso attributo. 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, avrei le possibili combinazioni dei valori dei due attributi. 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.

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

Nota. Where di solito è usata prima di Group by

HAVING

Poi c'è la clausola **having**, che seleziona solo alcuni gruppi. Se vogliamo selezionare solo i gruppi (ocio) che soddisfano un determinato predicato, possiamo usare la clausola having. 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
```

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
```

Nota: c'è differenza sostanziale tra having e where: mentre **having seleziona dei gruppi, where 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 da 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.

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)
```

19/3/18

Cambiato schema logico che ho stampato
Ha aggiornato anche create.sql

Per fare più commenti in SQL basta fare /* testo */

Ha modificato anche il database:

Mutate: se id del volo sta nell'insieme degli id che vialo la chaive gli assegno NA, altrimenti gli assegno tailnum.

```
# 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)t
```

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
```

Conclusione → I vincoli sono tutti soddisfatti.
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 (union,iIntersect, except)

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)

In SELECT con UNION non considera i duplicati; glielo posso imporre col comando

```
-- 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
```

Mentre le tabelle risultato di una interrogazione posso contenere righe duplicate, attenzione che 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.

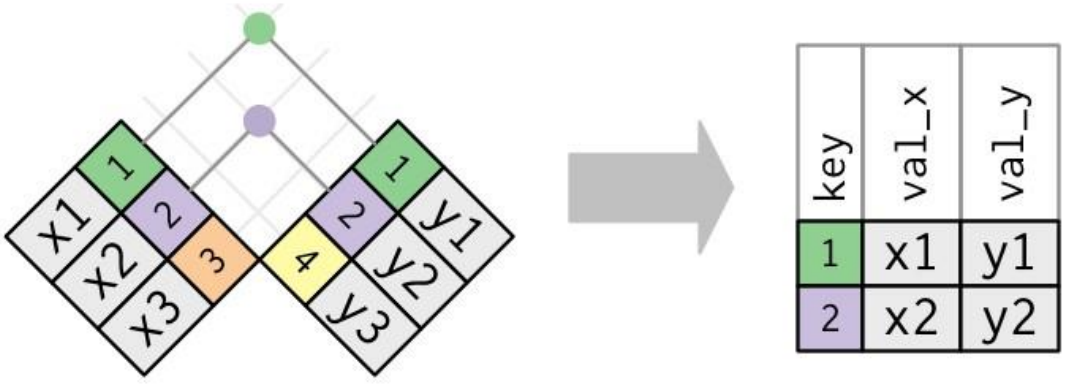
JOIN

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 informazione in tante tabelle quante sono le entità, legando queste tabelle tramite le parole esterne o straniere. Una volta che ho separato/normalizzato la mia informazione 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 *nxm* righe. Formalmente è questo il prodotto cartesiano:

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

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 AθB, dove A è un attributo di R, 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:

$$|AXB|=|A||B|$$

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, vogliamo 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 dà alcun risultato perché non ce ne sono, nessuna paura. Potrei trovare tutti i cammini arbitrari? Cioè di lunghezza arbitraria? Per esempio, si dà una città di partenza e trovano tutte le destinazioni che posso raggiungere da Roma con un arbitrario numero di scali. Ci servirebbe un'interrogazione infinita, vedremo una spiegazione formale che spiega perché 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 60
00 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 quello consigliato per scrivere il join, e ha alcuni vantaggi.

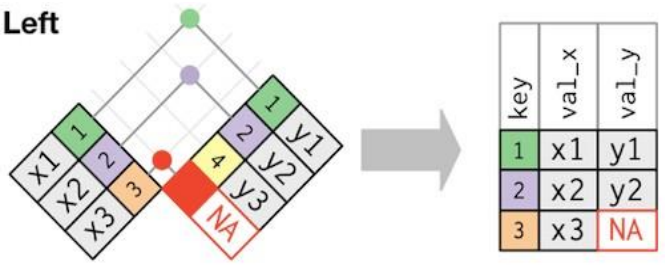
- È più chiaro, poiché separa la condizione di join da altre condizioni
- Permette di distinguere tra diversi tipi di join: inner join, ,left outer join, right outer join e full outer join.

INNER JOIN

filtra le tuple che soddisfano le condizioni di join (clausola usata fino ad ora). Seleziona solo la combinazione delle righe delle tabelle che soddisfa la combinazione boolena.

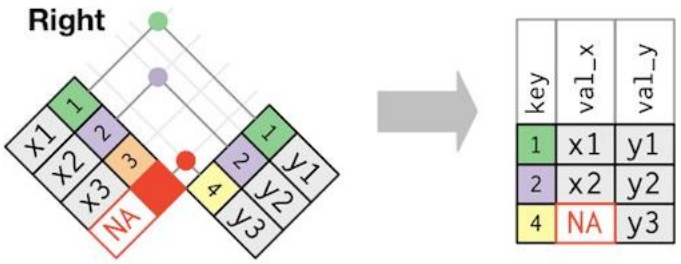
LEFT OUTER JOIN

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)



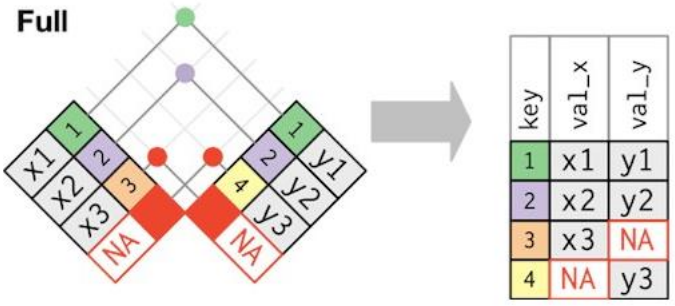
RIGHT OUTER JOIN

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)



FULL OUTER JOIN

unione delle due precedenti (left outer e right outer), cioè prende le tuple che soddisfano la condizione di join e tutte le tuple 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.

Right and full joins are not implemented in SQLite. ☹

Vediamo un esempio:

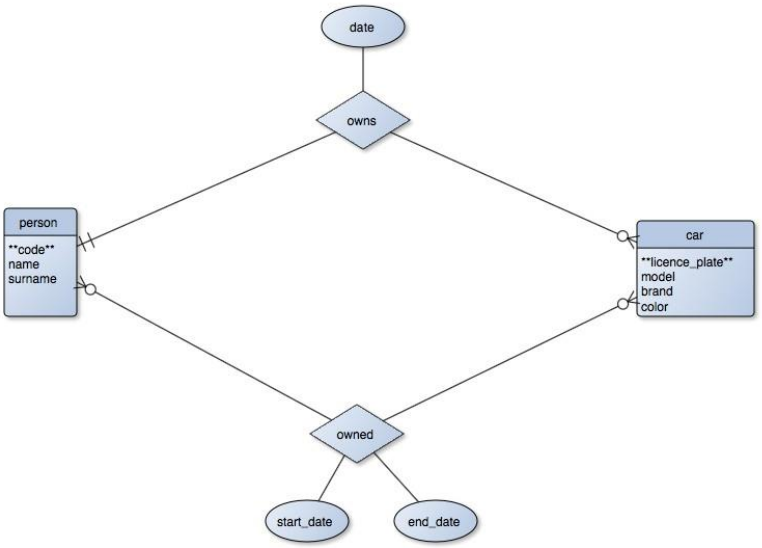
```
-- flights and the corresponding plane manufactured, including flights that have no match i
n planes (there are some)
select flights.id, flights.tailnum, planes.manufacturer
from flights left join planes on flights.tailnum = planes.tailnum

-- 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 join* e il *full join* 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.tailn
um
```

```
-- 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 = pla
nes.tailnum union all
 select flights.id, flights.tailnum, planes.manufacturer from flights left join planes on f
lights.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(tax_code, name, surname)
car(license_plate, model, brand, color, owner, date)
owned(person, car, start_date, end_date)
car(owner) –> person(tax_code)
owned(person) –> person(tax_code)
owned(car) –> car(license_plate)

ESERCITAZIONE CONCLUSIVA SU SQL

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;
```

```
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);
```

20.3.2018

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) o
r ":memory:" (for a in-memory database). This database will be automatically delet
ed 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), de
st = 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 fligh
ts 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 risult
ato è una tabella. Qui una query è una stringa di R, quindi si scrive tra virgolet
te.

# query with SQL
query1 =
"select id, month, day, sched_dep_time, carrier, flight number
from flights
```

```
where month = 12 and day = 25
limit 10"

bGetQuery(nyc, query1)

query2 = "select flights.id, airports2.alt, airports1.alt, (airports2.alt - airpor
ts1.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 q
uery 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 p
riori 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(r
s)) {  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.

1.8 make SQL from R {Rmd}

Esercizio in Rstudio

http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/make/SQL_from_R/SQL_from_R.Rmd

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 x e divisione / nei Reali 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 dà 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 una 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.

1.9 Algebra Relazionale e Calcoli

Introdurremo in modo un po' più formale il modello relazionale, 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 basi di dati non sono solo uno strumento, un programma o un software, ma sono una teoria. Vedremo un accenno di questa teoria.

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

$$schema(R) = \{A_1, \ldots, A_m\}$$

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 schema(R) = {A1, . . . , Am} cartesiano: è un elemento del prodotto cartesiano:

$$D(A1) \times D(A2) \times \ldots \times D(Am)$$

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.

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. Una **query relazionale** è una combinazione di un numero finito di operatori dell'algebra relazionale. 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 r_2;

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

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.
# 5 DL      Delta Air Lines 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 *select* 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 dà 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 connessione logica Booleana: $\wedge (and), \vee (or), \neg (not)$ 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 $\models$ F, è definita ricorrentemente, come segue:

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

La selezione, applicata ad una relazione se uno schema di relazione R rispettando una formula di 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 i 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 tra i connettivi logici Boolenai $\wedge, \vee, \neg$ presenti in una formula di 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 schema di relazione R con schema(R)= schema (R1) ∪ schema (R2) 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

A	B	C

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	B	C	D

1. Cos'è $r_1 \bowtie r_2$ se $schema(R_1) = schema(R_2)$? L'intersezione
2. Cos'è $r_1 \bowtie r_2$ se $schema(R_1) \cap schema(R_2) = \emptyset$? 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 $schema(R)$ e B un attributo non nello $schema(R)$. La rinomina, ρ, di A in B nella relazione r, è una relazione su uno schema di relazioni S, dove $schema(S) = (schema(R) \setminus \{A\}) \cup \{B\}$, definita da:

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

Utile quando faccio JOIN su 2 tabelle (relazioni) che hanno 2 nomi diversi:

ρ(r,B=A) "rho di R dove A viene rinominato al posto in 'B'". La relazione avrà argomento 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 e 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 albero 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.

https://dbis-uibk.github.io/relax/

ESERCIZIO

1. il numero del volo, il numero di coda dell'aereo e il costruttore aereo di voli fabbricati da EMBRAER. Ricorda che le tabelle dei voli e degli aerei condividono l'attributo dell'anno con semantica diversa.
Devo rinominare l'attributo in comune YEAR alle due tabelle, non voglio fare il join su quelle due tabelle

σ(planes,manufacturer=EMBRAER)>π({tailnum})

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

2. il numero del volo, il mese, il giorno, l'ora, l'origine e la visibilità dei voli decollati con una visibilità di 3 miglia

(flights⋈σ(ρ(planes,yearPlanes=year),manufacturer=EMBRAER))>π(
{flight,tailnum,manufacturer})

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

3. traiettorie di volo di lunghezza 2 (X -> Y -> Z)

ρ(π(flights,{origin,dest}),{X=origin,Y=dest})⋈
ρ(π(flights,{origin,dest}),{Y=origin,Z=dest})

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

4. traiettorie di volo di lunghezza 3 (X -> Y -> W -> Z).

ρ(π(flights,{origin,dest}),{X=origin,Y=dest})⋈
ρ(π(flights,{origin,dest}),{Y=origin,W=dest})⋈
ρ(π(flights,{origin,dest}),{W=origin,Z=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. Puoi generalizzare a percorsi di lunghezza arbitraria?

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 quantificatori $\forall$, $\exists$. Vedremo un parallelismo allora tra 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 $\mathscr{R} = \{R_1, R_2, \ldots, R_m\}$ 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 $R \in \mathscr{R}$ 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 $\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n | F(x_1, x_2, .., x_n)\}$ 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:

$$flights1 \cap flights2$$

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

PROIEZIONE π

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
```

Vogliamo recuperare, cioè far la proiezione di month, day e flight. Nelle tabelle precedenti 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.

Selection σ

(tralascia la seconda formula)

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 $\bowtie$

Scrivo la variabile in entrambe le tabelle (faccio join naturale su x5 carrier)

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:

$$flightsNarrow \bowtie airline$$

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

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

Esercizio

```
(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
```

```
(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
```

Trovare cammini lunghezza 2: (X –> Y –>w–> 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?
In pratca non ha soluzione la query 'dammi tutti i cammini da x a y in un numero finito di passi'(?) L'insieme di risposte di una query di algebra relazionale è sempre finito, poiché tutte le operazioni operano su relazioni finite. Tuttavia, questo non vale per il calcolo relazionale.

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 : A | ¬P (x)}

da l'insieme costituito da numeri maggiori di dieci e lo zero, quindi infinito. Mentre {x : a , y : B | P (x) ∧ 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. *se xRy allora xR*y*;
2. *se xRw e xR*y allora xR*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:

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

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, xRy = "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).

Quindi preso un insieme di formule, una teoria, se trovo un modello per ogni sua sottoteoria allora anche la teoria ha un modello. E viceversa. Detto altrimenti se tuti i sottomodelli sono soddisfacibili, allora è soddisfacibile anche la teoria.

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), $\emptyset -> A$, 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

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 $F \models X -> attr(R)$.

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. $F \models X -> A$. Cioè insieme degli attributi che sono determinati funzionalmente da X.

Teorema: $F \models X \rightarrow Y$ 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 $Studend(S), Course(C) -> Room(R)$ 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. $r = \bowtie_{i=1}^{K} \pi_{R_i}(r)$ dove $R_1 \ldots R_k$ è decomposizione di R. Questa proprietà è detta **lossless join** (senza perdita di informazione:

Esempio:

$R(\underline{A}, B, C, D)$

$\mathscr{F} = \{A -> B, C -> D\}$

Creiamo due relazioni $R_1(\underline{A}, B)$ 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 $\bigcup_{i=1}^{k} \mathscr{F}_i = \mathscr{F}$ ($\mathscr{F}_i$ è l'insieme delle dipendenze $X -> Y$ in $\mathscr{F}$ tali che $XY \subseteq R_i$). Allora si dice che è **fedele** o preserva le dipendenze.

Esempio: $R(\underline{A}, B, C)$, $\mathscr{F} = \{A -> B, B -> C, C -> A\}$ 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 allora se $B -> C$ e $C -> A$ allora $B -> \underline{A}$, 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_1

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

1.10 Invited speaker: Nicola Vitacolonna on data normalization {pdf}

26/3/18

CONFERENZA DI VITACOLONNA (argomento: basi di dati –penunltimo argomento)

NORMALIZZAZIONE

ESEMPIO
Modella esami universitari

Studente S, course C, chapter, H, Time T, Room R
Non conviene usare un unico schema → problema di ridondandza, quindi problemi di anomalie\ incosistenze.
Anche se in R avrei un solo dataframe.

Nozione DIPENDENZA FUNZIONALE: "X → Y" "X determina (funzionalmente) Y. con X, Y insiemi di attributi. Esprime vincolo di integrità. Stabilisce che ogni volt che 2 tuple coincidono per certi attributi in X, allora devono coincidere anche per gli altri attributi in Y.

Nozione di **SODDISFACiBILITA'**:
R schema relazionale con d f X→ Y
r istanza di R
R soddisfa X→Y sse per ogni t, t' \in r t[x]=t'[x] ➔ t[A]=t'[A]

Le dipendenze funzionali sono date assieme allo schema. Immagino che ci sia un tabellone con un solo valore per cella.

Studente	Corso	Capitolo	Time	Room

S C→ T
S T → R
T R → C
[S C → R]

Nozione di **CONSEGUENZA LOGICA**: un insieme di dipendenze logiche ne implica un'altra. Cioè
Ogni istanza che soddisfa F soddisfa X→ Y SSE F (pigreco rovesicato) X→Y.

Esempio su quaderno
F={A→B, B→C} (transitività)
Se ho 2 coppie di tuple che coincidono sui valore dell'attributo A che devono coincidere con valori attributo B, allora vale la transitività per valori su attributo C.)

(faccio inferenza)
A→B, B→C
A→ C

X→ YZ posso togliere attributi sul alto destro, e la conseguenza continua a valere
X→ Y

Index 1

X→Y
XZ→Y

Riflessività
XY→Y

Regola dell'aumento
Posso aggiungere attributi sul alto dx e sx
X—Y
XZ→YZ

Il concetto di CHIAVE in termini di DIPENDEZA FUNZIONALE

SUPERCHIAVE SSE F (pi greco rovesciato) X → attr(R)
Una chiave è una superchiave minimale

(date tutte le edipendenze funzioanli di uno schema esistono algoritmi che mi permettono di determinare tutte le ?chiavi ?o superchiavi. Index2

CHIUSURA di X rispetto ad F corsivo
È l'insieme delgli attributi che sono determinati funzionalmente da X rispetto ad F corsivo

Teorema
(*)Trasforma condizione semantica in un forma algoritmica

È algoritmo iterativo

- Cerco dipendeze funzionali in cui lato dx è incluso
- Contiene attributi dx più tutti gli attributi che stanno in AV
- Cerco tt le diepndenza (scrivo qll a dx)
- Cerco a sx di → lettere a dx e scrivo roba a dx

Regola euristica per determinare le chiavi:

- Se attributo compare solo a dx nelle diepndenze funzionali, non determina funzionalmenteniente → non può far parte di nessuna chaive. Attribuit lati sx: no chiavi
- Attributi che stanno solo a sx e mai a dx, devono neccessariamente far parte di tutte le chiavi delle relazioni. Attributi lato sx sempre in ogni chaive.

Sintesi.

- Dipendenza funzionale
  Definizione 2.5 (Conseguenza logica): Dato uno schema $(R, \mathcal{F})$ e data una dipendenza funzionale $X \to Y$ su R, diciamo che $\mathcal{F}$ ha come conseguenza logica $X \to Y$ (o, equivalentemente, diciamo che $\mathcal{F}$ implica $X \to Y$), in simboli $\mathcal{F} \models X \to Y$
- Soddisfacibilità
- Chiusura
  Teorema della chiusura): $\mathcal{F} \models X \to Y$ se e solo se $Y \subseteq X_+^{\mathcal{F}}$

Proprietù di una decomposizione

1) Data una decomposizione $\rho = \{R1 , \dots , Rk \}$ di $(R, \mathcal{F})$ diciamo che ρ è senza perdita d'informazione (o anche che ρ ha una lossless join o che ρ è lossless) se, per ogni istanza r di R che soddisfi $\mathcal{F}$ si ha $r = \pi\text{attr}(R1)(r) \bowtie \pi\text{attr}(R2)(r) \bowtie \cdots \bowtie \pi_{\text{attr}(Rk)}(r)$.

2) Diciamo che una decomposizione $\rho = \{(R1 , \dots , Rk)\}$ di $(R, \mathcal{F})$ preserva le dipendenze (o anche, che è fedele) se e solo se $k \cup i{=}1\ \mathcal{F}i \equiv \mathcal{F}$.

Esempio 3.9: Dato lo schema R(A, B, C) con dipendenze $\mathcal{F} = \{A \to B, B \to C, C \to A\}$, si consideri la decomposizione $\rho = \{R1(A, B), R2(B, C)\}$.
Pg 8

La normalizzazione in BCNF (Boyce-Codd Normal form)
Fino pg 12

\