

A.A. 2021/22

# FONDAMENTI DI SISTEMI INFORMATIVI

PROF. ALBERTO BELUSSI

FABS (CON IL CONTRIBUTO DI ELISA ZANELLA : )

## NOTA

**Questi appunti/sbardinatura/versione “discorsiva” delle slides sono per mia utilità personale,** quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

Il simbolo [✓] nel titolo indica che il capitolo è stato revisionato.

Appunti e puttanate sono scritte *principalmente* in questo stile.

**Questo file fa parte della mia collezione di sbobinature,  
che è disponibile (e modificabile!) insieme ad altre in questa repo:  
<https://github.com/fabfabretti/sboninamento-seriale-uniVR>**

## INDICE

NOTA .....	1
Indice .....	2
Introduzione – modello relazionale .....	3
Calcolo relazionale [✓].....	5
Esercitazione: calcolo relazionale [✓].....	9
Transazioni: ripasso [X].....	11
Nuove tecnologie: evoluzione dei database [✓].....	12
Big Data e Hadoop [✓].....	15
Data design [✓].....	24
Design concettuale in UML [✓] .....	25
Esercitazione: modellazione in UML [✓].....	26
MAPPING FISICO: HBase [✓].....	27
Sharding e amazon[✓].....	31
MAPPING FISICO: DynamoDB [✓] .....	36
Document databases [✓] .....	40
Document database CouchDB: linguaggi N1QL[✓] e mongodb [X].....	42
Analisi dei linguaggi di interrogazione dei sistemi document BASED [X].....	44
MAPPING FISICO: DocumentDB [✓] .....	47
Columnar databases [✓] .....	50
Database a grafo [✓].....	53
INTERNALS: distributed pattern [✓] .....	55
INTERNALS: consistency models [✓].....	60
Database spaziali: fondamento teorico [✓] .....	64
DATABASE SPAZIALI: Geo-RDBMS [✓] .....	69
DATABASE TEMPORALI .....	74
Dipendenze funzionali e forme normali [X].....	78

## INTRODUZIONE – MODELLO RELAZIONALE

È molto studiato, ha una base teorica solida, è ancora moltousato e anche le nuove tecnologie tendono a dare interfacce simili in quanto è noto a tutti.

L'approccio del modello relazionale ha la caratteristica principale di basarsi su un **approccio/paradigma dichiarativo**: è il sistema che si occupa di **stabilire come trovare il piano di esecuzione**, mentre io devo solo preoccuparmi di descrivere correttamente il risultato. ! Questo non è più vero con le nuove tecnologie: sono io a dover specificare e trovare il modo migliore di generare il risultato !

Questo approccio è *molto comodo* 😊

L'approccio dichiarativo è usato sia per la definizione dei dati che anche nella generazione della struttura: escluso il problema della ridondanza, in generale non mi preoccupo di come la tabella sarà partizionata e memorizzata nel file system. ! Nei sistemi nuovi, di nuovo, sono io a dover essere cosciente di come memorizzo i dati, e ho il controllo fino all'implementazione fisica. !

### Ripassino: modello relazionale

#### Modello

Si intende l'insieme degli **strumenti formali** che mi consentono di descrivere la struttura e le proprietà dei dati implementati su uno specifico sistema.

→ Diverso dalla concezione generica del termine; per noi l'implementazione in uno specifico contesto è lo **schema**.

Abbiamo a disposizione:

- **Tabella/relazione**
- **Domini di base**: prelievo i valori che miconsentono di descrivere le proprietà e che istanzio nelle righe; sono tipicamente atomici – aka non esiste un'ulteriore struttura, e di solito sono stringhe, date, numeri naturali, etc.
- **Vincoli di integrità**: i più importanti dal punto di vista strutturale sono primary key e vincoli di integrità referenziale.

Possiamo usare il linguaggio per:

- Interrogare la DB inserire
- Definire le strutture

Noi focalizziamo sull'interrogazione, che usa **l'algebra relazionale**, e SQL – che è una sintassi alternativa del **calcolo relazionale**. Il calcolo relazionale ci permetterà di classificare la capacità espressiva dei sistemi.

#### Definizioni formali

<i>Insiemi di base</i>	Per dare una definizione più precisa al modello relazionale (che ci tornerà utile per il calcolo relazionale), ripartiamo introducendo alcuni insiemi di simboli di base che ci servono per rappresentare la struttura della base di dati e i valori da inserirvi. <ul style="list-style-type: none"><li>• <b>DOM</b> è l'insieme dei possibili valori che vengono usati nelle varie istanze di informazione (=uniamo tutti i domini per semplificarcici la vita)</li><li>• <b>ATTNAME</b> è l'insieme dei possibili nomi delle colonne/attributo</li><li>• <b>RELNAME</b> è l'insieme di nomi per le tabelle/relazioni.</li></ul>
<i>Funzione <math>SORT(R)</math></i>	Ciascuna relazione ha un nome $R \in RELNAME$ e un insieme finito di attributi, dato dalla funzione $SORT$ . $SORT : RELNAME \rightarrow \mathcal{P}(ATTNAME)$ <i>Esempio</i> $R(U) \circ R(A_1 \dots A_n) \rightarrow SORT(R) = U = \{A_1 \dots A_n\}$
<i>Arità di una relazione</i>	L'arità della relazione R è il numero degli attributi. $ARITY(R) =  SORT(R) $ <i>Esempio</i> $SORT(R) = \{A_1 \dots A_n\} \rightarrow ARITY(R) = 3$
<i>Tuple di una relazione</i>	Definiamo il contenuto di una relazione attraverso le tuple. In realtà ci sono due modi, questo è il più usato. Una tupla di $R(U)$ è una funzione $t : U \rightarrow DOM \text{ con } U \text{ finito e } U \subset ATTNAME$

	<p>Ovvero specifica una relazione fra una relazione e i valori di una riga (?)</p> <p>È un numero finito di coppie, perché U (insieme di attributi di una tabella) è finito, ed è un sottoinsieme di ATTNAMEx.</p> <p><i>Esempio</i></p> $U = \{cf, surname, date, dob\}$ $t[cf] = \text{CMB...}, t(surname) = \text{Combi}, \dots$ <p>C'è anche un modo diverso di rappresentare le righe come ennupla, ovvero la prospettiva dove non usiamo nomi per etichettare le colonne e ci affidiamo all'ordine.</p> <p><i>Esempio</i></p> $t = < \text{CMB...}, "Carlo", \dots >$ $t[1] = \text{CMB...}, t[2] = "Combi", \dots$
<i>Schema relazionale</i>	Lo schema di una relazione è composto dal suo nome seguito dalla lista dei nomi degli attributi: $R(U) o R(A_1 \dots A_n)$
<i>Schema di un database</i>	È un set finito e non vuoto di schema di relazioni $R_{cors.} = \{R_1(U_1) \dots R_n(U_n)\}$
<i>Istanza di una relazione</i>	È un set finito di tuple di $SORT(R)$ : $I(R) = \{t_1 \dots t_n\} \text{ con } t_i: SORT(R) \rightarrow DOM \ (1 \leq i \leq n)$
<i>Istanza di un database</i>	L'istanza di un database su uno schema $\mathcal{R} = \{R_1(U_1) \dots R_n(U_n)\}$ è l'insieme di tutte le relazioni istanza dello schema relazionale che appartengono a $\mathcal{R}$ : $I(\mathcal{R}) = \{I(R_1) \dots I(R_n)\}$ (È l'unione delle istanze delle relazioni, quindi è l'insieme delle $I(R)$ )

# CALCOLO RELAZIONALE [✓]

## Introduzione

- È il **fondamento teorico** dell'SQL, ed è il modo in cui rappresento le interrogazioni per fare analisi della potenza espressiva.
- È **indipendente dalla tecnologia** (= non è SQL) ed è **dichiarativo**.
- È una teoria logica del primo ordine.

Usiamo la versione chiamata "calcolo relazionale sui domini". Esiste anche quella sulle tuple, su cui si basa più SQL, ma preferisce mostrarcì questa.

### Esempio di espressione

Restituire il nome e cognome dello studente con matricola indicata

$$\{s, n \mid \exists c, d (\text{STUDENT}(c, s, n, d) \wedge c = 'VR110908')\}$$

→ Produciamo nel risultato le ennuple che sostituite nella formula che segue la rendono vera, prendendo i valori nel dominio di base. Questa è anche l'intuizione per capire se è scritta correttamente :)

!! I valori vengono da DOM, non dalla base di dati: devo considerare anche valori che non ci sono !!

Per definire una teoria logica, mi serve definire una sintassi e una semantica.

- **Sintassi:** stabilisco
  - Alfabeto: i simboli e
  - Grammatica: come compongo i simboli per costruire frasi sintatticamente corrette.
- **Semantica:** stabilisce come assegnare un significato ai simboli; devo dire
  - Basic domains che considero per dare valori alle variabili delle espressioni
  - Interpretazione dei simboli
  - Interpretazione delle formule

## Sintassi

### Alfabeto

- Costanti:  $\text{CONSTNAME} = \{c_1, c_2 \dots\}$
- Variabili:  $\text{VARNAME} = \{x_1, x_2 \dots\}$
- Nomi delle relazioni:  $\text{RELNAME} = \{R_1, R_2 \dots\}$
- Operatori logici binari:  $\leftrightarrow, \rightarrow, \wedge, \vee$
- Operatori logici unari:  $\neg$
- Simboli di confronto:  $=, \neq, <, >, \leq, \geq$
- Quantificatori:  $\exists, \forall$
- Simboli ausiliari:  $\{\}, (), | \dots$

### Grammatica

Struttura base di una query:

- $\text{QUERY} \rightarrow \{\text{VARS} \mid \text{FORMULA}\} \mid \{\text{FORMULA}\}$
- $\text{VARS} \rightarrow \text{var} \mid \text{var}, \text{VARS} \text{ con } \text{var} \in \text{VARNAME}$
- $\text{FORMULA} \rightarrow \text{ATOMIC} \mid \text{QUANTIFIER}$
- $\text{FORMULA} \rightarrow (\text{FORMULA}) \mid \text{LOGICFORMULA}$

Tipo	Formula	Esempi
------	---------	--------

<b>Formula atomica</b>	Base	$ATOMIC \rightarrow pred(TERMS)$ con $pred \in RELNAME$ $TERMS \rightarrow TERM, TERMS$ $TERM \rightarrow var \mid const$ con $var \in VARNAME, const \in CONSTNAME$	$TERMS : x_1, c_1, c_2$ $ATOMIC : R_1(x_1, c_1, c_2)$
	Theta: confronto una variabile con un termine	$THETA \rightarrow var = TERM \mid var \neq TERM$ $\mid var < TERM \mid var \leq TERM$ $\mid var > TERM \mid var \geq TERM$	$x_1 > c_1$
<b>Formula non atomica</b>	Con quantificatori	$QUANTIFIER \rightarrow \exists VARS. FORMULA \mid \forall VARS. FORMULA$	$\exists x_1, x_2. (R_1(x_1, x_2))$
	Con operatori logici	$LOGICFORMULA \rightarrow FORMULA \wedge FORMULA$ $\mid FORMULA \vee FORMULA$ $\mid FORMULA \leftrightarrow FORMULA$ $\mid FORMULA \rightarrow FORMULA$ $LOGICFORMULA \rightarrow \neq FORMULA$	$R_1(x_1, x_2) \wedge x_1 > 1$

Esempi:

$$\begin{aligned} & \forall x_1. (\exists x_2. (R_1(x_1, x_2) \vee (R_2(x_2) \wedge x_2 > 10))) \\ & R_3(x_1, x_2) \wedge x_2 = 'pippo' \\ & R_3(x_1, x_2) \vee \exists x_4. (R_4(x_4, x_2) \wedge x_2 > 0) \end{aligned}$$

## Variabili free e bound

### Def. Free variable

Data una formula  $\varphi$ , una variabile  $x$  che compare nella formula è detta **variabile libera** se  $x$  non è usata da nessun quantificatore. → libere in quanto libere di assumere valori!

### Def. Bound variable

Data una formula  $\varphi$ , una variabile  $x$  che compare nella formula è detta **variabile legata** se  $x$  è usata come variabile quantificata in una formula che usa un quantificatore.

! Non c'è una regola che impone di usare nomi diversi per variabili libere e legate, tuttavia sarebbe meglio seguirla. !

## Semantica di una formula

Partiamo dal fatto che stiamo definendo una logica del primo ordine; la logica ci dà strumenti per definire un'interpretazione.

### Interpretazione in logica

In ambito logico, si dice **interpretazione** una corrispondenza che assegna a ciascun simbolo un significato.

L'interpretazione è composta da:

- Insieme  $\Delta^I$  che rappresenta il **dominio dell'interpretazione**, ovvero l'**insieme da cui pescano i simboli**. Per noi è il dominio dei valori da cui pescano i dati che rappresento nelle colonne delle tabelle. Più in generale, per semplicità, lo immaginiamo come l'insieme degli interi. Potenzialmente, potrei interpretare le stesse formule in domini diversi e ottenere risultati diversi.
- **Corrispondenza** tra i **simboli di costante usati** e il **dominio** indicato sopra (es. che il simbolo 0 rappresenta il numero 0).
- **Corrispondenza** tra gli **operatori di confronto** θ e le **relazioni di confronto** nel dominio.
- **Corrispondenza** tra i **simboli delle relazioni** n-arie.

### Interpretazione per la semantica della nostra formula

- Il dominio di interpretazione  $\Delta^I$  corrisponde al dominio  $DOM$  che contiene tutti i valori che popolano le tuple. Per semplicità, consideriamo il dominio degli interi  $DOM = \mathbb{Z}$
- Le costanti  $CONSTNAME$  corrispondono agli elementi di  $I$
- Gli **operatori di confronto** corrispondono alle **relazioni d'ordine e uguaglianza** degli interi

- I **simboli di predicato** di *RElname* si mappano sulle **istanze delle relazioni** del database  $\{I(R_1) \dots I(R_n)\}$
- Gli **operatori** logici hanno la semantica definita dalle tabelle di verità.
- I **quantificatori** hanno la seguente semantica: la verità di quella formula dipende da come andiamo ad analizzare i valori che possono assumere le variabili quantificate.  
L'idea generale è che ho una formula con un certo insieme di **variabili quantificate**.
  - Le variabili quantificate **prendono dal dominio di interpretazione**
  - $\forall x. \phi(x)$  è **vera** se per ogni valore  $v_x \in \Delta^I$  la formula è **vera**
  - $\exists x. \phi(x)$  è **vera** se esiste almeno un valore  $v_x \in \Delta^I$  la formula è **vera**.

### Assegnamento di una variabile

Questo formalismo piace ai logici ma noi lo usiamo solo per dire quello che vogliamo cit. Introduciamo una funzione  $\alpha$  che semplicemente **sostituisce alla variabile un valore**.

#### *Def Funzione di sostituzione $\alpha$*

Esiste una **funzione di sostituzione**  $\alpha: VAR \rightarrow \Delta^I$ , ovvero la  $\alpha$  mi sostituisce tutte le variabili libere con dei valori, al fine di avere una formula interpretabile.

#### *Def Modello*

Infine, si dice che la coppia  $I, \alpha$  è il **modello** per  $\varphi$  se la formula risulta vera.

### Semantica della query

La sintassi della query è

$$Q = \{\vec{x} | \varphi\}$$

con  $\vec{x} = (v_1 \dots v_n) = FV(\varphi)$ , **vettore di variabili che è esattamente identico all'insieme delle variabili libere** della formula.

Esempi di query booleane:

- $\varphi \equiv \forall x. R(x, x)$
- $\varphi \equiv \forall x. \exists y. R(x, y)$

Quindi,  $\{x_1, x_2 | R_3(x_1, x_2, x_3)\}$  non è corretta perché ho  $x_3$  libero! Posso correggere con  $\{x_1, x_2 | \exists x_3. R_3(x_1, x_2, x_3)\}$

### Semantica della query in RC

Per dire il risultato della interrogazione Q, definiamo qualche altra cosuccia:

#### *Def Tupla di costanti*

Dato un vettore di variabili  $\vec{x}$  e un assegnamento  $\alpha : VARNAME \rightarrow \Delta^I$ , allora  $\alpha(\vec{x})$  è la **tupla di costanti** che si ottengono **sostituendo ciascuna variabile  $v_i$  del vettore con la costante  $\alpha(v_i)$** .

#### *Def Valutazione di una query Q su I*

Data  $Q = \{\vec{x} | \varphi\}$  una query in calcolo relazionale, allora il **risultato di Q su I è definito come**  $Q(I) = \{\alpha(\vec{x}) | I, \alpha \models \varphi\}$

Ovvero l'**insieme di tutte le sostituzioni**  $\alpha(\vec{x})$  tali che  $I, \alpha$  è un modello per  $\varphi$  – ovvero che se interpreto la formula con  $I, \alpha$  essa è soddisfatta.

Ovviamente, questo **non sarà poi il procedimento per generare il risultato della query** – aka non faccio il test su tutte le possibili sostituzioni delle variabili – ma è quello che funziona dal punto di vista della semantica.

! Da qui bisognerebbe classificare le diverse espressioni del calcolo, e alcune non rappresentano interrogazioni; hanno un comportamento “strano” e andranno escluse – sono le **espressioni dipendenti dal dominio**. Ci sono meccanismi sintattici che permettono di escluderle, e a quel punto il calcolo relazionale su domini safe è equivalente all'algebra relazionale. !

## Esempi

Consideriamo il database schema:

cd = code  
 PATIENT(cd,na,sa): na = name  
 sa = surname

pa = patient  
 (codice del paziente che manifesta il sintomo)  
 SYMPTOM(pa,sn,in,en): sn = nome del sintomo  
 in = data inizio  
 en = data fine

*Q1: cognome dei pazienti contenuti nella base di dati*

Anonimamente:  $\{x \mid \exists y, z. (\text{PATIENT}(y, z, x))\}$

→ dato un certo x, questo x va nel risultato se esistono altri due valori y e z tali per cui con x completano una tupla di paziente.

Più efficacemente:  $\{\text{surname} \mid \exists \text{code}, \text{name}. (\text{PATIENT}(\text{code}, \text{name}, \text{surname}))\}$

*Q2: cognome dei pazienti che hanno almeno un sintomo*

Ovvero che esiste una riga nella tabella SYMPTOM che riguarda quel paziente.

$\{\text{surname} \mid \exists \text{cd}, \text{na}. (\text{PATIENT}(\text{cd}, \text{na}, \text{surname}) \wedge \exists \text{sn}, \text{in}, \text{en}. \text{SYMPTOM}(\text{cd}, \text{sn}, \text{in}, \text{en}))\}$

- [!] Tento di capire come procedere in senso ordinato. Innanzitutto voglio che il risultato sia un cognome, ovvero legare la variabile risultato alla tabella paziente e completare la tupla con l'esistenziale delle variabili che mancano.  
 → **Esiste un codice e un nome che messi vicino al cognome costruiscono una tupla di paziente.**  
**Esiste un paziente con quel cognome.**
- [!] Potrei anche chiudere qui lo scope di queste due variabili, in generale. Ma in questo caso no: perché voglio che anche la seconda parte della condizione riguardi quel paziente.  
 → **La condivisione di variabili fra predici è tipicamente un meccanismo per fare la join.**

! Non seguendo strettamente questo modo di ragionare possiamo generare mostri cit. !

*Q3: scrivere una query che trova nome e cognome dei pazienti che hanno manifestato almeno due sintomi diversi.*

*Ussignur.* Voglio i pazienti per cui esistono due righe diverse nella tabella sintomo e che sono diverse fra loro.

$\{\text{name}, \text{surname} \mid \exists \text{cd}, \text{sn}_1, \text{sn}_2, \text{in}_1, \text{in}_2, \text{en}_1, \text{en}_2.$   
 $\quad \text{PATIENT}(\text{cd}, \text{name}, \text{surname})$   
 $\quad \wedge \text{SYMPTOM}(\text{cd}, \text{sn}_1, \text{in}_1, \text{en}_1)$   
 $\quad \wedge \text{SYMPTOM}(\text{cd}, \text{sn}_2, \text{in}_2, \text{en}_2)$   
 $\quad \wedge \text{sn}_1 \neq \text{sn}_2\}$

- Mettiamo tutte le variabili quantificate all'inizio; potenzialmente potrei anche metterle man mano.,
- Cerco un paziente: quindi la prima cosa che metto è il fatto che deve esistere la tupla voluta in paziente.
- Devono esistere due entry con quel paziente in symptom, diverse fra loro.

*Q4: scrivere una query in calcolo relazione per trovare il codice dei pazienti che hanno manifestato tutti i loro sintomi dopo il 1 maggio 2015.*

$\{\text{code} \mid \exists \text{na}, \text{sa}. (\text{PATIENT}(\text{code}, \text{na}, \text{sa}) \wedge \forall \text{sn}, \text{in}, \text{en}. (\text{SYMPTOM}(\text{code}, \text{sn}, \text{in}, \text{en}) \rightarrow \text{in} > '2015'))\}$

- Di nuovo, come prima cosa agganciamo il risultato con l'esiste
- Cerco tutti i sintomi del paziente e verifico che la condizione sia soddisfatta per il codice fissato: epr ogni sostituzione che completa quella tupla, allora deve soddisfare.  
 [!] Metto un'implicazione e non un and!! Con l'and, sarebbe come dire "qualsiasi data deve essere  $\geq$ " e quindi esce sempre falso.  
**Con l'implicazione, mostro che la seconda parte è significativa SOLO QUANDO LA PRIMA è VERA. IL QUANTIFICATORE UNIVERSALE DEVE LAVORARE TUTTE LE VOLTE CHE TROVO EFFETTIVAMENTE UN SINTOMO; TUTTE LE ALTRE SOSTITUZIONI DEVONO RENDERE VERE LA FORMULA, PERCHÉ non mi interessano!**

## ESERCITAZIONE: CALCOLO RELAZIONALE [✓]

### Database

#### Exams

- MEDICAL\_CENTER(CenterCode, Name, Municipality, Region, Lat, Long);
- TEST(Date, Patient, Type, Time, Processed, Result, Center)
- PATIENT(PatCode, Age, Nationality, DateOfBirth, ResMunicipality, ResRegion)

Notice that there exist the following referential integrity constraints:

*TEST.Center* → *MEDICAL\_CENTER* and  
*TEST.Patient* → *PATIENT*. Moreover, the attribute *Processed* contains (*true*, *false*) and  
*TEST.Type* ∈ {‘molecularswab’, ‘rapidtest’}

### Query

Q1: trova i pazienti di età > 55 e nazionalità “irish” o “english” riportando nel risultato codice, resmunicipality e nazionalità del paziente.

{patcode, resmunicipality, nationality | ∃ age, dob, resregion. (PATIENT(patcode, age, nationality, dob, resmunicipality, resregion)  
 $\wedge age > 55$   
 $\wedge (nationality = 'irish' \vee$   
 $nationality = 'english')$  }

Q2: centri in veneto locati non a Verona né a Venezia, riportando codice nome e coordinate del centro.

{centercode, name, lat, long | ∃ municipality, region. (MEDICAL\_CENTER(centercode, name, municipality, ‘veneto’, lat, long)  
 $\wedge municipality <> 'Verona'$   $\wedge municipality <$   
 $> 'Venezia'$  ) }

- Posso mettere ‘veneto’ direttamente nella tupla, anziché aggiungere  $\wedge resregion = 'veneto'$

Q3: test dei pazienti che vengono dalla sicilia, riportando codice paziente risultato e data del test.

{patcode, result, date | ∃ ... . (TEST(date, patcode, type, time, processed, result, center)  
 $\wedge PATIENT(\text{patcode}, \text{age}, \text{nationality}, \text{dob}, \text{resmunicipality}, \text{resregion})$   
 $\wedge resregion = 'sicily'$  }

- Uso lo stesso patcode in più relazioni per esprimere la join.

Q4: Pazienti che hanno fatto sia test rapido che molecolare in un giorno. Nel risultato: nazionalità e data.

{ nationality, date | ∃ patcode, age, dob, resmunicipality, resregion. (PATIENT(patcode, age, nationality, dob, resmunicipality, resregion)  
 $\wedge \exists type_1, time_1, processed_1, result_1, center_1 . (TEST(date, patcode, type_1, time_1, processed_1, result_1, center_1))$   
 $\wedge \exists type_2, time_2, processed_2, result_2, center_2 . (TEST(date, patcode, type_2, time_2, processed_2, result_2, center_2))$   
 $\wedge type_1 = 'molecular'$   $\wedge type_2 = 'rapid'$   
 $)$   
 $}$

- Uso *patcode* per fare la join.
- Di nuovo, potevo anche mettere *molecular* e *rapid* direttamente nelle tuple.

Q5: Centri che il 1 gennaio 2021 non hanno fatto esami di nessun tipo, riportando nome e municipalità del centro.

{ name, municipality | ∃ centercode, region, lat, long. (MEDICAL\_CENTER(centercode, name, municipality, region, lat, long)  
 $\wedge \neg \exists date, patient, type, time, processed, result, centercode . ($   
 $TEST(date, patient, type, time, processed, result)$   
 $)$   
 $)$   
 $}$

Q6: trova nome e municipality dei centri che hanno fatto almeno un test ciascun giorno di July 2020.

```
{ name, municipality | ∃ centercode, region, lat, long. (
    MEDICAL_CENTER(centercode, name, municipality, region, lat, long)
    ∧ ∀d. (
        CALENDAR(d) ∧ d ≥ '31/07/2021' ∧ d ≤ '31/07/2021'
        → ∃ patient, type, time, processed, result . (
            TEST(d, patient, type, time, processed, result, centercode)
            )
    )
}
```

Q7: trova il più giovane paziente che ha fatto un test di tipo 'rapid test' in agosto 2020, riportanto codice e età del paziente.

```
{code, age | ∃ name, db, m, r . (PATIENT (code, age, n, db, m, r)
    ∧ ∃ d, tm, p, r, ct. (TEST(d, c, 'rapidtest', tm, p, r, ct) ∧ d ≥ 1/aug/20 ∧ ... d ...)
    ∧ ¬∃ c', a', n', db', m', r'. (
        PATIENT(c', a', n', db', m', r')
        ∧ db' > db
        ∧ ∃ d', tm', p', r', ct'. (
            TEST(d', c', 'rapidtest', tm', p', r', ct')
            ∧ d' ≥ '1/aug/20' ∧ d' ≤ '31/aug/20')
        )
    )
}
```

Q8: Trova per ciascun paziente se esiste la coppia di test positivi consecutivi di tipo "molecular swab", riportando il codice del paziente e la data di entrambi i tests. (use only the attribute "Date" for ordering the tests on the temporal axis).

```
{ n | ∃ c, m, r, lt, lg. (MEDICAL_CENTER(c, n, m, r, lt, lg)
    ∧ ∃ d, p, tm, p, r. (
        TEST(d, p, ty, tm, p, r, c)
        ∧ ∃ a, na, db, rm. (
            PATIENT(p, a, na, db, rm, 'Liguria')
            ∧ ¬∃ c1, n1, m1, r1, lt1, lg1. (
                MEDICAL_CENTER(c1, n1, m1, r1, lt1, lg1)
                ∧ c ≠ c1
                ∧ ∃ d1, p1, tm1, r1. (
                    TEST(d1, p1, ty1, tm1, p1, r1, c1)
                    ∧ ∃ a1, na1, db1, rm1. (
                        PATIENT(p1, a1, na1, db1, rm1, 'liguria')
                        ∧ d1 > d
                    )
                )
            )
        )
    )
)}
```

Q9: trova il centro che ha fatto l'ultimo test a un paziente dalla liguria.

```
{n, c, p1, ty1, tm1, r1 | ∃ m, r, lt, lg. (MEDICAL_CENTER (c, n, m, r, lt, llg)
    ∧ ¬∃ p, ty, tm, pr, r. (
        TEST('2021/jan/1', p, ty, tm, pr, r, c)
    )
    ∧ ∃ pr1. (
        TEST('2021/may/21', p1, ty1, tm1, pr1, r1, c)
    )
)}
```

## Tassonomia delle query in calcolo relazionale

---

Consiste nella classificazione delle interrogazioni.

### C1: Single collection

La query riguarda una singola collezione, ovvero si esprime con un solo predicato agganciato a una tabella. Nei sistemi nuovi che hanno struttura complessa, già queste sono molto significative e coprono molte delle esigenze.

$$\{x_1 \dots x_n | \exists y_1 \dots y_m. (R(x_1 \dots x_n, y_1 \dots y_m) \wedge f(x_1 \dots x_n, y_1 \dots y_m))\}$$

### C2: Part-whole

Part-whole significa che ci sono due collezioni W e P, dove W è una collezione di wholes e P di parts. P corrisponde a una tabella che contiene come foreign key una primary key della tabella corrispondente a W.

La vogliamo caratterizzare perché è quella che mi porta ad avere un disegno fisico sui sistemi nuovi, che rappresenteranno spesso insieme queste classi (= es. un insegnamento insieme ai suoi moduli). Se nei relazionali devo rappresentare tabelle separate, nei sistemi nuovi spesso questo è più facile a patto di avere tutto assieme a livello fisico.

$$\begin{aligned} & \{w_1 \dots w_n, p_1 \dots p_k | \exists y_1 \dots y_m. (W(w_1 \dots w_n, y_1 \dots y_m) \wedge f(w_1 \dots w_n, y_1 \dots y_m) \\ & \quad \wedge \exists z_1 \dots z_h. P(w_1 \dots w_n, p_1 \dots p_k, z_1 \dots z_h) \wedge f(w_1 \dots w_n, p_1 \dots p_k, z_1 \dots z_h))\} \end{aligned}$$

### C3: Part-whole con quantificatore universale

È come sopra ma ho un per ogni o un esiste negato.

$$\begin{aligned} & \{w_1 \dots w_n, p_1 \dots p_k | \exists y_1 \dots y_m. (W(w_1 \dots w_n, y_1 \dots y_m) \wedge f(w_1 \dots w_n, y_1 \dots y_m) \\ & \quad \wedge \neg \exists z_1 \dots z_h. (P(w_1 \dots w_n, p_1 \dots p_k, z_1 \dots z_h) \wedge f(w_1 \dots w_n, p_1 \dots p_k, z_1 \dots z_h)))\} \end{aligned}$$

Con w attributi della chiave esportata che collegano P a W.

### C4: Relationship based di primo tipo

Immagino che fra le due classi E e F ci sia una maggiore indipendenza; sono due collezioni indipendenti che hanno una chiave esportata.

$$\begin{aligned} & \{w_1 \dots w_n, p_1 \dots p_k | \exists y_1 \dots y_m. (E(w_1 \dots w_n, y_1 \dots y_m) \wedge f(w_1 \dots w_n, y_1 \dots y_m) \\ & \quad \wedge \exists z_1 \dots z_h. (F(v_1 \dots v_n, p_1 \dots p_k, z_1 \dots z_h) \wedge f(w_1 \dots w_n, p_1 \dots p_k, z_1 \dots z_h))\} \end{aligned}$$

Con v attributi che rappresentano la chiave esportata di F in E.

### C5: Relationship based di secondo tipo

In questo caso la query ha una relazione molti a molti fra le classi. Ho bisogno di tre predicati per agganciare tre tavole. Tipicamente i sistemi nuovi non sono in grado di fare queste cose.

### C6: Relationship based di primo o secondo tipo con quantificatore universale

Autoesploratorio. I sistemi non lo fanno e fin.

## TRANSAZIONI: RIPASSO [X]

Scusate ma foldo. Non ho tempo e conto sul fatto che sia la roba fatta in triennale. Non che me la ricordi eh, ma sono arrivata con l'acqua alla gola :') Feel free to chiedermi il DOCX e aggiungere!

## NUOVE TECNOLOGIE: EVOLUZIONE DEI DATABASE [✓]

Vedremo i nuovi requisiti e le nuove rivoluzioni.

### Prima rivoluzione dei database (1950-'60)

Riguarda l'avvento delle prime tecnologie dei DBMS; fino a quel punto i dati non erano memorizzati in modo strutturato, poiché venivano usati solo in ambito di ricerca. Non appena la necessità di avere database uscì dai laboratori, ecco che diventa importante l'analisi e la gestione dei dati (con un aumento di dimensione) e nascono i primi sistemi dedicati ai dati:

- **Network model** : soluzione integrata
- **Hierarchical model** : proposto da IBM

Questi sistemi si basano ancora su **schede perforate**.

### Seconda rivoluzione dei database (1970-'90)

Prima non c'era una modellizzazione e implementazione ben precisa; con l'avvento del **modello relazionale** si formalizza effettivamente la teoria delle basi di dati, inizialmente implementato in **System R**. Poco dopo entrano anche altri modelli, come Oracle, e viene definito lo standard SQL.

- **Si separa completamente la rappresentazione fisica dei dati da quella logica**, introducendo un livello di astrazione che permette di avere un'interfaccia gestibile.
- Vengono introdotte le **transazioni** con le loro **proprietà**.

I sistemi relazionali hanno fatto da padrone per molti anni, che hanno soppiantato qualsiasi altra tecnologia per più di trent'anni.

### Terza rivoluzione dei database (2005 – oggi)

È il **primo attacco al modello relazionale**, a causa di alcune novità messe in campo da alcuni attori centrali del mercato mondiale.

#### Google

Il primo attore a mettere in crisi il relazionale è Google; esso ha la **necessità di gestire il suo motore di ricerca**, che aveva bisogno di rappresentare e indicizzare in maniera compatta tutto il web. Questa esigenza, sintetizzabile in "volume e velocità", ha portato Google ad abbandonare quasi subito le soluzioni relazionali poiché il **volume** dei dati da trattare era troppo elevato per fornire le prestazioni necessarie.

Inoltre, il sistema relazionale è pensato per dati che vengono continuamente **modificati**; a Google questo non interessa, poiché vuole un insieme di dati di cui non gli interessa chi ha fatto la modifica and stuff (non sono i requisiti diversi dalle banche)

Tutto questo fa sì che Google produca una soluzione completamente nuova: si riparte **rivedendo l'intera architettura**, partendo dal file system.

- **Non c'è più un server unico**: si passa ad avere clusters (insieme di server e macchine che condividono dati e lavorano insieme), e viene introdotto il GFS (Google File System).
- Si introduce anche un **paradigma di esecuzione parallela** degli algoritmi su questi dati, detto **MapReduce**, e introduce un database non relazionale distribuito chiamato BigTable.

Questa tecnologia non era accessibile a tutti; Yahoo inizia un progetto opensource che mette queste tecnologie a disposizione di tutti, ovvero **Hadoop**.

#### Amazon

Amazon aveva la necessità di **gestire il suo ecommerce online**, e quindi aveva bisogno delle transazioni ma con un insieme di clienti provenienti dall'intero globo. Questo chiaramente è un salto notevole di requisiti, e i sistemi relazionali vanno in crisi; lo stesso succede ai social network del periodo, quali Facebook.

All'inizio si cerca di dare risposte a questa esigenza anche attraverso il **relazionale**: Oracle produce **Oracle-RAC**, un tentativo di produrre sistemi in grado di scalare orizzontalmente, ma senza successo.

Di conseguenza Amazon e altri iniziano a **destrutturare il relazionale e definire nuove architetture**.

#### Sharding

in questo caso si decide di dividere orizzontalmente i dati e farli gestire a nodi/server separati.

- Si tende a **suddividere il dato in orizzontale** (es. avendo una collezione di clienti con le loro informazioni, che nel relazionale sarebbero in tabelle magari diverse, qui vanno tutte sullo stesso nodo in modo che se voglio recuperare le info di un cliente posso farlo da un solo nodo).
- Lo si fa attraverso un **grande cluster di macchine** Linux con sopra mySQL (per evitare le pesanti e complicate soluzioni Oracle): si gestiscono loro i dati dividendoli in maniera intelligente.

È chiaro che dividere l'informazione in migliaia di nodi in questo modo ci crea dei problemi: abbiamo un approccio relazionale, **ma non tutte le operazioni previste sono ancora fattibili**: è evidente che lavorando sul singolo nodo è ancora tutto a disposizione, ma non posso pensare di fare alcune operazioni in modo massivo su tanti nodi, poiché è pesantissimo (es. join fra info su nodi diversi).

Inoltre, le **proprietà ACIDE sono perse fra nodi diversi**: lo sharding non evita la replicazione dell'informazione, e garantire le proprietà fra diversi nodi è *complicato* e pesante.

Quindi, amazon propone la sua soluzione (!= mysql) con **DynamoDB**, il primo database key-value che garantisce alcune proprietà sulla consistenza.

#### Cloud computing

Il fatto che si rendano disponibili alcuni servizi di calcolo – come amazon AWS – con l'idea **"infrastructure as a service"** va stretto al relazionale, perché esso **scala poco orizzontalmente** e non si adatta. Anche questa è una situazione nuova, che porta in crisi il modello relazionale che non è più preferibile e viene sostituito da soluzioni che scalano meglio quali DynamoDB e SimpleDB.

#### Document databases

Nascono dal fatto che il modello relazionale ha un certo **"mismatch" rispetto al paradigma di programmazione ad oggetti**.

Vengono introdotti alcuni sistemi basati su **object-relational mapping**, che cercano di mappare meglio verso il relazionale, ma non sono sufficienti – in particolare quando si adotta lo stile di programmazione **AJAX** che consiste nel trasferire in modo sincrono dati dal server verso le pagine dell'applicazione dati attraverso XML (e successivamente JSON). Nascono quindi soluzioni per gestire direttamente i dati JSON, come CouchDB e MongoDB.

Da qui nascono i **sistemi document-based**, ovvero dati semistrutturati simili a **XML** ma che per la sua verbosità solitamente sono rappresentati in **JSON**. Questi sistemi hanno proprio l'obiettivo di rappresentare l'informazione come **documenti semistrutturati**, in istanze molto più complesse e **nidificate**.

#### NewSQL

C'è ancora un fronte aperto: intorno al 2007, StoneBraker (ideatore di Postgres) pubblica un articolo dove suggerisce una revisione a causa del cambio delle architetture hardware, rendendo falsi alcuni presupposti del relazionale. Nascono quindi sistemi **H-store** e **C-store**, che si svilupperanno negli **"in-memory distributed databases"**, aka soluzioni che **non prevedono di avere a che fare con la memoria secondaria**: caricano tutto in memoria e in qualche modo, soprattutto nei sistemi cluster, ciò è possibile e anche vantaggioso. Un esempio è il sistema **Spark**.

Questo fa decadere alcune cose fondamentali del sistema relazionale.

Inoltre, c'è il **nuovo approccio** di vedere il database organizzato sempre come tabelle, ma invece di organizzare l'info per renderla fruibile dalle righe (=dalle istanze), si passa dalle **colonne** – aka hanno una **strutturazione fisica che rende più semplice lavorare sulle operazioni sulle colonne** (count, media...)

#### Esplosione del NoSQL

C'è stata persino una crisi a inizio anni 2000 😊

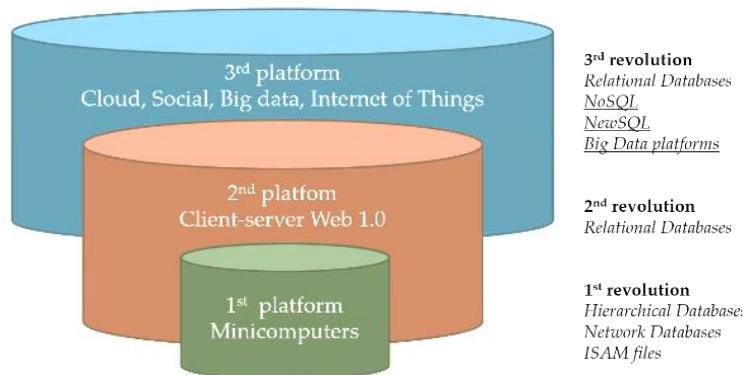
Negli anni 2008-2009 nasce una nuova classe di database detta **Distributed Non-Relational Database management Systems**, **NoSQL**, aka **Not Only SQL**, che fondamentalmente propongono **sistemi distribuiti** basati non sul modello relazionale per gestire questo modello.

Alcuni sopravvivono ancora oggi: **MongoDB**, **Cassandra**, **HBase**.

L'idea è che, se vogliamo, siamo in un momento dove non c'è una soluzione per tutto: bisogna scegliere, e questo rende tutto più difficile: i sistemi nuovi non sono per niente piatti (es. non tutti gli accessi hanno lo stesso costo), per cui vengono privilegiati alcuni accessi e IMPOSSIBILITATI altri – sbagliare DB potrebbe voler dire che dovrò ricominciare da caso a un certo punto.

## Riassuntino

- **Prima rivoluzione:** conseguenza dell'emersione dei pc
  - Mi devo concentrare sul design fisico
  - I dati sono legati al singolo sistema
  - Solo i programmati hanno accesso ai dati; non ho interfacce ad alto livello
- **Seconda rivoluzione:**
  - C'è un design logico indipendente da quello fisico
  - Ho un linguaggio più flessibile, che permette anche ad altri oltre che ai programmati di accedere
  - Gestisco anche la corrispondenza logico-fisico, indicizzazione...
  - Tanti strati di astrazioni che permettono di estendere in tanti sensi, frutto di 30 anni di lavoro e dominio del mercato
- **Terza rivoluzione:** non c'è più una sola architettura fondante.
  - Ho nuovi requisiti dati da social network, IoT che portano il RDBMS al punto di rottura
  - Non basta più il relazionale, ma in molti casi resta ancora la tecnologia giusta (es. applicazioni "tradizionali")



## BIG DATA E HADOOP [✓]

Partendo dall'esperienza di Google, cerchiamo di definire le questioni del dato. Il dato diventa molto grande: la tecnologia precedente non era sufficiente, e in qualche modo qui abbiamo un passaggio in più:

- **Più dati:** non sono le singole organizzazioni devono gestire i dati del proprio core business, ma ci sono anche una serie di dati generati in modo nuovo:
  - Mondo social
  - Dati generati dai sensori e dalle macchine
  - Transazioni, come quelle degli ecommerce
- **Più effetti:** prima i dati gestivano i processi delle aziende, e una volta completato il processo "rimanevano lì", inutilizzati. Invece qui diventano importanti anche le tecniche di ripescaggio e analisi dei dati collezionati.

Queste info aggiuntive possono alimentare sistemi predittivi, che consentono di fare **turbopredizioni**.

La vera rivoluzione "industriale" dei dati è che prima i dati erano generati dai processi interni aziendali, mentre da qui in poi i dati vengono anche **dall'esterno** – la mole di dati è generata dalla rete, che si fa raccoglitore di tutto – sensori, social network, enti pubblici e statistica, meteo... Inoltre, è un problema che **si autoalimenta**.

Da qui il problema del big data. Google inizia a gestire questi big data con un approccio tradizionale attraverso una **collezione di server**, ma ovviamente non ha più retto.

Quindi ne ha proposta una completa nuova, creando un nuovo hardware: un **cluster**, il **Google Modular Data center**, che è un insieme di macchine molto grossi con forte parallelismo.

Insieme a questo si propone anche di usare un nuovo metodo di elaborazione il **MapReduce**.



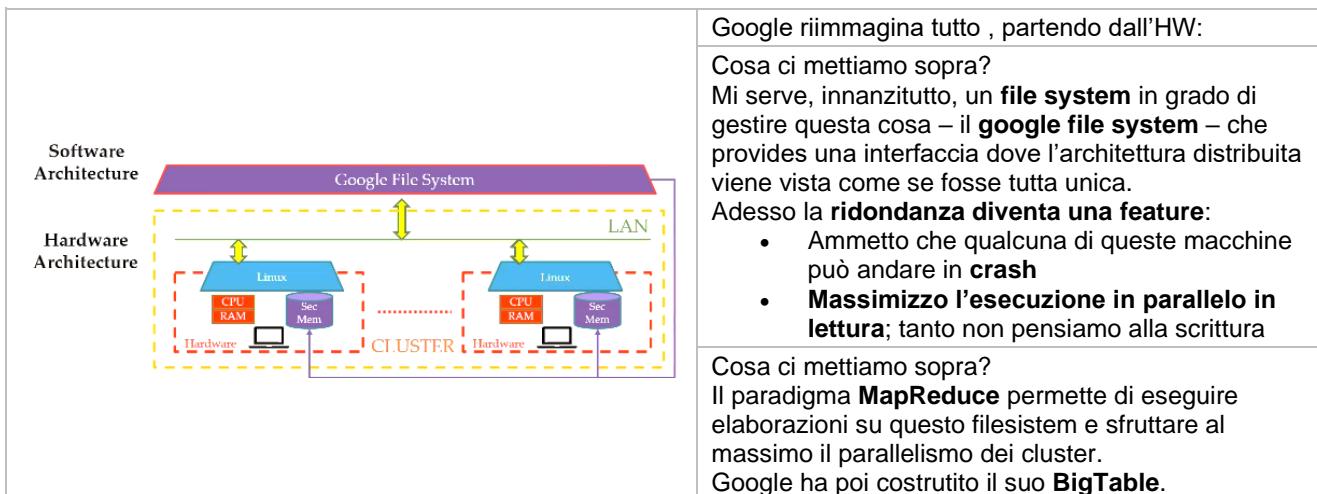
### Scalabilità di RDBMS

Scalare = adeguarsi mano a mano a richieste che crescono, sia in termine di dati che in termine di query.

Ci sono due metodi:

<b>Scalare verticalmente</b> <p><b>Aumento le risorse</b>, tipo CPU RAM e memoria secondaria, rendendo il server più potente, ma resta sempre solo una macchina. È quello che si è sempre adottato.</p>	
<b>Scalare verticalmente 2</b> <p>Aumentiamo la memoria secondaria <b>esternalizzando completamente su un altro hardware la gestione della memoria secondaria</b>. Anche questa non risponde alle richieste.</p>	
<b>Scalare orizzontalmente</b> <p>Aggiungo servers, ma c'è qualcuno fuori (il resto del DBMS) che gestisce questo insieme di macchine. Immagino sempre un'esecuzione centralizzata che poi viene "indirizzata" alle macchine giuste per il recupero dei dati. Ho addirittura dei protocolli per garantire le proprietà globali delle transazioni</p>	

## Soluzione di Google



Google riimmagina tutto , partendo dall'HW:

Cosa ci mettiamo sopra?

Mi serve, innanzitutto, un **file system** in grado di gestire questa cosa – il **google file system** – che provides una interfaccia dove l'architettura distribuita viene vista come se fosse tutta unica.

Adesso la **ridondanza diventa una feature**:

- Ammetto che qualcuna di queste macchine può andare in **crash**
- **Massimizzo l'esecuzione in parallelo in lettura**; tanto non pensiamo alla scrittura

Cosa ci mettiamo sopra?

Il paradigma **MapReduce** permette di eseguire elaborazioni su questo filesistem e sfruttare al massimo il parallelismo dei cluster.

Google ha poi costruito il suo **BigTable**.

## Hadoop

L a versione open source di questo sistema Google è Hadoop. Le caratteristiche sono:

- **Scalabilità**: sia rispetto all'architettura (aggiungere nuovi cluster)
- Grande **capacità di immagazzinamento** dei dati, con scalabilità orizzontale
- Grande **affidabilità**, affidata alla replicazione dell'informazione sui vari nodi
- **Scalabilità del processing** grazie all'approccio mapreduce
- **Schema on the read**: flessibilità sullo schema dei dati, che non è definito a priori

Hadoop propone:

- Il proprio system file distribuito, che deriva direttamente da quello di Google e si chiama **HDFS** (Hadoop Distributed File System).
  - Immaginiamo di avere tanti nodi di macchine linux semplici; ciascun nodo del cluster, organizzato in rack e collegato da una rete locale; ciascun nodo è sia un gestore di un certo insieme di dati che l'esecutore di una porzione di lavoro.
- Sopra l'HDFS troviamo il framework **MapReduce**, come già visto.
- Sopra ancora c'è un database per la gestione dei dati, che è il primo sistema NoSQL definito in maniera precisa e chiamato **HBase**.

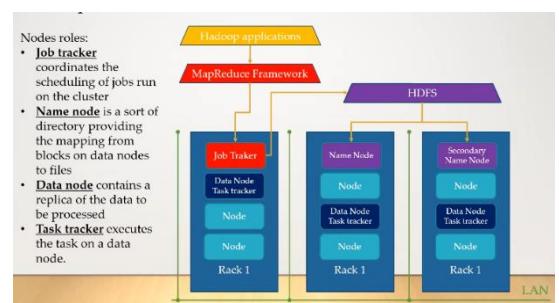
### Architettura Hadoop 1.0

Considerando la prima versione, abbiamo i nodi del cluster che ospitano sia **dati che processi di elaborazione** – partono tutti come **data node** , e in caso di job diventano anche **task tracker**. Esistono dei nodi secondari che possono sostituire in caso di guasto.

Il sistema HDFS ha bisogno di nodi che gestiscano il file system, conoscendo come i file sono organizzati e partitionati sul sistema. Questi nodi supportano l'accesso al file system in modo trasparente.

Esistono anche repliche di questi nodi, sempre per l'idea della reliability per replicazione.

Quando eseguo un job, uno dei nodi diventa esecutore del job e responsabile di coe i vari task sono eseguiti e rimessi insieme. Nelle versioni successive di Hadoop questa parte è gestita da moduli più avanzati, ma comunque con questa idea di fondo. Non c'è traccia di indici, transazioni o interrogazioni 😊



### Architettura Hadoop 2.0

Viene introdotto un modulo più avanzato (**YARN**) per la gestione di risorse che permette di eseguire in parallelo più task mapreduce. YARN si è ancora evoluto, ma insomma si capisce come pian piano si aggiungono i mattoncini che ci danno una soluzione via via più siile ai sistemi tradizionali.

Questo nuovo modulo può gestire anche **framework di esecuzione diversi da MapReduce**.

## MapReduce

Per elaborare i dati devo scrivere un programma che gestisca il contenuto dei file attraverso il framework MapReduce, che rende parallela l'esecuzione della mia elaborazione. Il modello è molto semplice: i file sono **collezioni di record** strutturate come **chiave-valore**.

Record = **mattoncino** (!!!) da cui parte il modello; è una coppia **<k,v>**. Potrei rappresentare un **sistema relazionale** con questo sistema mettendo le righe della tabella come valore. Alla fine nel 90% dei casi si fa ciò perché la gente è abituata così!!!!!!

Il file è una collezione di record; accedo ai files con il programma mapreduce. Il **modello mapreduce appartiene alla famiglia "keyvalue"**. Il framework mapreduce introduce una modalità di computazione che prevede due tipi di task, eseguiti in due fasi distinte:

- **Fase di mapping:** vengono eseguiti i map tasks
- **Reduce task:** vengono eseguiti uno o più tasks di reduce

L'input è diviso in chunks chiamati split, e ciascuno è assegnato a una map task che elabora e produce un risultato. I risultati vengono poi messi assieme e dati in input alla fase di reduce. I nostri task vengono poi associati ai vari nodi del cluster dal sistema stesso, ed eseguiti sui dati assegnati sullo split di quel map. Tipicamente si hanno job con un solo reduce, e il reduce riceve il risultato di tutti i map senza aspettare che finiscano tutti. È un'esecuzione tipicamente batch.

### Esempio

Vogliamo un programma per insieme di dati che rappresentano il tempo meteorologico, rilevati da stazioni piazzate in tutto il globo. Immaginiamo che i nostri files contengano queste info e ogni record habbia la forma . Ho un file per ogni anno e località. Voglio trovare la temperatura più alta per ogni anno contenuto.

#### 1. Decisione di come rappresentare in key-value.

Se non ho una chiave significativa per l'elaborazione da fare, metto una chiave fittizia. Qui non ne ho; quindi il mio input sarà

- Key** : offset, ovvero posizione dall'inizio del file
- Value** : il record stesso

#### 2. Mapping.

Spesso serve a buttare via i valori non corretti dalla computazione ed estrarre quello che mi serve; produco un dato che contiene solo l'essenziale.

Mi serve solo anno e temperatura. Le coppie generate vanno messe assieme, da una fase detta shuffle, che praticamente fa un ordinamento or smth

#### 3. Reduce.

La fase di reduce prende i risultati e li mette insieme – per esempio, calcola l'effettivo massimo.

In codice java = è lo schifo.

<b>Map</b> <pre> import java.io.IOException; import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.LongWritable; import org.apache.hadoop.mapreduce.Mapper; import org.apache.hadoop.mapreduce.Mapper; public class MaxTemperatureMapper     extends Mapper&lt;LongWritable, Text, Text, IntWritable&gt; {     private static final int MISSING = 9999;     @Override     public void map(LongWritable key, Text value, Context context)         throws IOException, InterruptedException {         String line = value.toString();         String year = line.substring(15, 19);         int airTemperature;         if (line.charAt(0) == '+') { // parseInt doesn't like leading plus signs             airTemperature = Integer.parseInt(line.substring(0, 2));         } else {             airTemperature = Integer.parseInt(line.substring(0, 2));         }         String quality = line.substring(92, 93);         if (airTemperature != MISSING &amp; quality.matches("[01459]")) {             context.write(new Text(year), new IntWritable(airTemperature));         }     } } </pre>	<p>Estendiamo una classe map e riscriviamo il metodo map, che prende in input una chiave e un valore, e un contesto.</p> <ul style="list-style-type: none"> <li>• Converti in stringa</li> <li>• Estraggo il valore</li> <li>• Controllo la stringa</li> </ul> <p>Se tutto torna, con la write scrivo in output la coppia <i>&lt;anno,temperatura&gt;</i>.</p>
<b>Reduce</b>	<p>Lo shuffle è implicito. Anche qui ho una classe e metodo reduce da scrivere; qui viene ricevuta una lista iterabile di valore.</p> <p>Il reduce, semplicemente, calcola il massimo.</p>

```

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}

Main
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class MaxTemperature {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }
        Job job = new Job();
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max temperature");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

Prepara il job: definisce la classe del job, alcuni parametri di input e output, la classe del map e del reduce, e infine il tipo di chiave risultato e del valore risultato.

C'è un forte giro sulla memoria secondaria. Però d'altra parte in memoria centrale non ci sta tutto...

*Come viene eseguito il job?*

È YARN a decidere come, ordine e schedule.

- **Divisione dell'input.**

L'idea è che l'input è diviso in blocchi, chiamati SPLITS. Immaginiamo che i blocchi siano quelli conci ui ho diviso i files sul file system (in realtà potrebbero essere divisi ancora, ma nsomma)

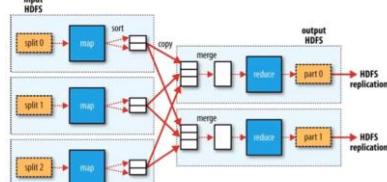
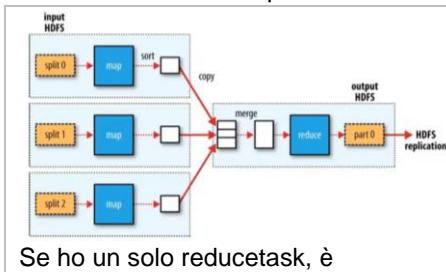
- Tipicamente, Hadoop assegna un map task a ogni split.
- La dimensione dello split è definita dalla configurazione di hadoop che sto facendo girare; di solito è 64-128 MB. Non è banale scegliere la migliore.

- **Assegnare il lavoro ai nodi.**

Sono già stati caricati e divisi, ma dove alloco il task di esecuzione? L'idea è che per minimizzare i task di esecuzione l'idea è di far eseguire il task di map sul nodo che ospita quello SPLIT. Di solito si riesce a ottenere ma non è sempre possibile; la località è un criterio di ottimizzazione.

- **Eseguire il maptask e il reducetask.**

I dati sono salvati in memoria secondaria, poi prelevati dallo shuffle, ordinati e portati sul nodo dove viene eseguito il reduce. Il risultato è poi salvato ancora sul file system hdfs. Quando i dati del mapper osno consumati dal reduce possono essere eliminati.



Se ho più reducetasks, allora la regola è che i record con la stessa chiave vanno dallo stesso reducer.

Hadoop non raggiungeva un gran risultato poiché solo le persone in grado di scrivere in mapreduce riuscivano a raggiungere i dati; la potenza espressiva è massima ma lo è anche la difficoltà. Serviva una soluzione più semplice con uno strumento di query per estrarre i dati. Sono dunque nate due soluzioni che introduscono un linguaggio di interrogazione: Hive e Pig. Hadoop, anche con questi nuovi strumenti, rimane un sistema per elaborazioni batch più che per query; la forma di questi due linguaggi trae dunque un po' in inganno, perché sembrano SQL ma quello che c'è dietro è ben diverso e non c'è un sistema di gestione di dati relazionale con le solite operazioni.

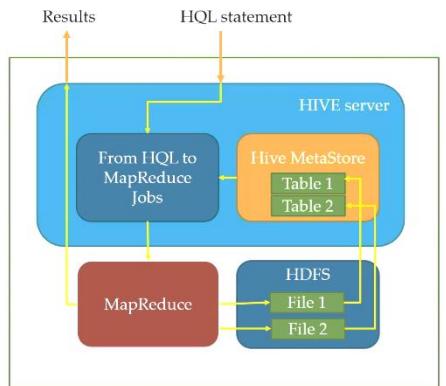
## HIVE

Hive è generalmente pensato come “**SQL per Hadoop**”, poiché assomiglia molto a SQL ed è molto semplice da usare per chi vi è abituato, ma dietro le quinte è molto diverso.

La prima cosa da dire è che il **modello di Hadoop resta key-value**. La corrispondenza è ovvia, perché basta mettere una chiave come chiave e il resto come value.

Ovviamente, non possiamo aspettare di eseguire query in realtime. Tuttavia, in seguito sono poi state implementate anche soluzioni per avere risposta immediata, ma sono sistemi diversi:

- IMPALA by Cloudera
- Integrazione di Hadoop con TEZ



## HQL

Simile a SQL, ha una clausola SELECT ... FROM ... WHERE

- È possibile fare JOIN nella FROM
- Inserire subqueries (che equivale a fare viste)
- È possibile fare query nidificate, ma:
  - Se ho operatori IN/NOT IN in query non possono esserci valori condivisi con la query esterna
  - Se ho operatori EXISTS/NOT EXISTS è obbligatorio avere valori condivisi con la query esterna.

Le differenze sono:

- HQL tenta di inserire qualcosa di non relazionalissimo, come **liste di valori rappresentate come array**
  - Explode() ritorna una riga per ogni elemento in un array
  - Json\_tuple() esplosione di un documento JSON
- C'è un'operazione estremamente difficile in questo caso: **le operazioni che devono scandire il tutto per il risultato**. Per esempio, infatti, non c'è l'ORDER BY.  
Quindi ci sono clausole diverse:
  - **SORT BY** ordina l'output **solo in ciascun reducer**
  - **DISTRIBUTE BY** richiede che la distribuzione degli output dai reducer non sia basata sull'hashing delle key ma sui nomi specificati nella clausola.
  - **DISTRIBUTED BY + SORT BY** può produrre ordinamento totale; si può abbreviare con **CLUSTER BY**

## PIG Latin

Qui il linguaggio è **procedurale** (e non dichiarativo). Date delle tabelle caricate su hadoop, genera l'elaborazione compilando l'espressione da PIG a MapReduce.

Il potere espressivo è simile, poiché PIG ottiene quanto ottiene HQL e forse qualcosa di più.

Le espressioni in PIG sono una sequenza di assegnamenti, dove a sinistra ho una nuova tabella e a destra le operazioni da applicare.

<b>LOAD</b>	<code>Students = LOAD '../Stud.rec' AS (Code: chararray, Name: chararray, BirthYear: int, Town: chararray, Degree: chararray);</code>	Abbiamo degli operatori esplicativi per caricare i dati, perché quello che si fa è caricare l'info, elaborarla e scaricarla – non serve a gestire la base di dati, ma solo ad analizzare i dati raccolti.
<b>DUMP</b>	<ul style="list-style-type: none"> <li>• <b>Tuple</b>: record semantics. (item1,item2,item3), ()</li> <li>• <b>Bag</b>: collection with duplicates. {code}{{tuple1),(tuple2)},{}{}</li> <li>• <b>Map</b>: set of key.value pairs. [key1#value1,key2#value2]</li> </ul>	Posso <b>incapsulare</b> cose o strutturare l'informazione (es. divido n di telefono in prefisso e il resto). Si usa poi la dot notation per navigare nella struttura.

<pre>Students = LOAD '../Stud.rec' AS     (Code: chararray, Name: chararray,      BirthYear: int, Degree: chararray,      Phone: tuple(prefix: chararray,                   number: chararray));</pre>	
<b>Selection</b>	
<pre>StudSEL = FILTER Students BY Phone.prefix='045'; DUMP StudSEL;</pre>	
Join	
<pre>StudWithExams = JOIN Students BY Code,                  Exams BY Scode; StudPROJ = FOREACH StudWithExams GENERATE Code,             Degree, Course, Grade; DUMP StudPROJ;</pre>	è il solito
Projection	
<pre>StudPROJ = FOREACH Students GENERATE             Name, Phone.prefix, Phone.number;</pre>	scandisco tutto il contenuto e genero un nuovo record prendendo alcuni dei valori che trovo dentro la struttura
<b>UNION</b>	
<pre>DegreeNames = FOREACH Students GENERATE Degree; CourseNames = FOREACH Exams GENERATE Course; Names = DegreeNames UNION CourseNames; DUMP Names;</pre>	
<b>Differenza</b>	
<pre>DegreeNames = FOREACH Students GENERATE Degree; CourseNames = FOREACH Exams GENERATE Course; JoinNames = JOIN DegreeNames LEFT BY Degree             CourseNames BY Course; DegreeNoCourse = FILTER JoinNames BY Course IS NULL; DegreeDiff = FOREACH DegreeNoCourse GENERATE Degree; DUMP DegreeDiff;</pre>	non esiste, ma posso usare il join esterno; uso un left join dove sicuramente ho tutte le istanze della prima collezione, e poi mantengo solamente quelle dove il corso è null
<b>ORDER BY</b>	
<pre>StudOrd = ORDER Students BY BirthYear; DUMP StudOrd; (83, Mary Green, 1964, Washington, Physics) (09, John Smith, 1970, London, Computer Science)</pre>	
<b>RANKING</b>	
<pre>StudRank = RANK Students BY BirthYear; DUMP StudRank; (1, 83, Mary Green, 1964, Washington, Physics) (2, 09, John Smith, 1970, London, Computer Science)</pre>	oltre all'ordinamento aggiunge con capo con la posizione in classifica.
<b>GROUP BY</b>	
<pre>(1964, { (83, Mary Green, 1964, Washington, Physics),           (21, Paul Esting, 1964, New York, Physics)}), (1970, { (09, John Smith, 1970, London, Computer Science) })</pre>	Come in SQL posso generare gruppi. Lo schema in risultato è fisso: ho una tabella group e un'altra con le istanze da cui sono partiti or smth.
<b>Count</b>	Posso unire group by e count per contare.

```

StudGroup = GROUP Students BY Degree;
StudCount = FOREACH StudGroup
    GENERATE group, COUNT(Students);
DUMP StudCount;

```

### Esempio

Per implementare lo stesso esempio visto prima con le temperature, questo sarebbe il risultato:

```

-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;

```

### Esempio a confronto: Pig Latin e HQL.

**Query:**  
count the number of customers for each country of Asia having more than 500 customers, reporting the name of the country and the number of customers.

**Pig Latin**

```

countrys = load 'PIG_COUNTRIES' as (country_id,
country_name, country_subregion, region);
customers = load 'PIG_CUSTOMERS' as (cust_id, first
name, last_name, postcode, city, country_id);
asianCnts = filter countrys by region matches 'Asia';
joined = join customers by country_id, asianCnts by
country_id;
grouped = group joined by country_name;
aggr = foreach grouped generate group,
COUNT(joined.customers::cust_id) as custCount;
moreThan500 = filter aggr by custCount > 500;
ordered = order moreThan500 by custCount desc
dump ordered;

```

### HQL

```

SELECT country_name, COUNT(cust_id) as custCount
FROM countries AS co
JOIN customers AS cu
ON (co.country_id = cu.country_id)
WHERE co.region='Asia'
GROUP BY country_name
HAVING COUNT(cust_id) > 500
CLUSTER BY custCount DESC

```

Poi tutto ciò viene tradotto in un job mapreduce.

## HBase

Sia HIVE che Pig consentono di accedere in maniera più semplice alla tecnologia Hadoop e al parallelismo di Map/reduce; però non si discostano più di tanto dal modello key-value a livello fisico, e per l'utente mostrano l'astrazione del database relazionale.

Al contrario, sopra HDFS e Hadoop è definito anche un sistema **completamente nuovo**: si tratta di HBase.

HBase è parente di Big Table di Google, ed è uno strumento per la gestione di grandi qtà di dati non memorizzabile nel relazionale proprio a causa della dimensione eccessiva. Offre un modello che astrae dalla struttura key-value che sta sotto, ma usa dei termini simili al relazionale; si parla di tabelle, benché sono completametne diverse.

La prima cosa che il sistema HBase eredita dall'approccio nuovo di Hadoop/HFS è il fatto che i dati sono rappresentati sul sistema sempre con almeno 3 copie. HBase usa questo meccanismo per proteggere i dati da eventuali guasti che potrebbero corrompere il contenuto; il fatto che ho delle repliche in HDFS garantisce che sia veramente remota la probabilità di perdere tutte e tre le copie dei dati.

Notiamo una differenza fondamentale: in Pig Latin tutte le istruzioni partono con un load, perché i dati NON sono già sul sistema. Tutte queste tecnologie ipotizzavano che i dati fossero su altri sistemi, quindi le “travasano” con mapreduce.

## Struttura

La tabella con righe e colonne, come nel relazionale, è ancora la struttura usata a livello fisico per i dati. Ma sono usate in modo un po' diverso:

- Esiste una **rowkey**: esiste una **chiave di accesso ai dati** ed è quella che memorizzo e caratterizza le righe. È **indicizzata**, ed è il principale modo di accedere ai dati.
- Le istanze di info sono rappresentate nelle righe; **ogni riga ha una chiave e una serie di famiglie di colonne**; ovvero, quando creo una tabella, ne definisco la chiave e una o più famiglie di colonne.
- Nelle famiglie di colonne ho le colonne effettive, e ogni riga può avere colonne diverse.
- I **nomi delle colonne sono dinamici** e possono cambiare; **ad ogni inserimento posso creare una nuova colonna**
- I valori sono inseriti nelle righe con un timestamp associato.
- Possiamo vedere ogni **column family** come una **mappa multidimensionale** dove inserisco valori associati a un nome di colonna e a un timestamp.
- I valori di una column family tipicamente sono vicini su memoria secondaria; ovvero, i dati di una stessa column family possono essere acceduti più velocemente.

## Esempi

Table USERS		PERSONAL DATA column Family	FRIENDS column Family						
Rowkey=Guy		<table border="1"> <tr><td>Name</td><td>Guy Harrison</td></tr> </table>	Name	Guy Harrison	<table border="1"> <tr><td>Jo</td><td>John</td></tr> <tr><td>jo@gmail.com</td><td>john@hotmail</td></tr> </table>	Jo	John	jo@gmail.com	john@hotmail
Name	Guy Harrison								
Jo	John								
jo@gmail.com	john@hotmail								
Rowkey=Jo		<table border="1"> <tr><td>Name</td><td>Joanna Fill</td></tr> </table>	Name	Joanna Fill	<table border="1"> <tr><td>John</td><td>Mary</td></tr> <tr><td>john@hotmail</td><td>mr@hotmail</td></tr> </table>	John	Mary	john@hotmail	mr@hotmail
Name	Joanna Fill								
John	Mary								
john@hotmail	mr@hotmail								

**! Su ogni istanza, per prima cosa devo vedere che attributi ha. Non è banale scrivere un codice del genere, perché per ogni istanza non so che attributi ci sono!!! Quindi avere un insieme di colonne fisse è una buona idea !**

- FRIENDS: il nome della colonna è un dato, il nome dell'amico! E come valore ho l'email.

Table SENSOR		LOCATION column Family	READINGS column Family						
Rowkey='S01'		<table border="1"> <tr><td>Coord</td></tr> <tr><td>Timestamp= 12334 value: 10.1 value: 45.54, 10.17</td></tr> </table>	Coord	Timestamp= 12334 value: 10.1 value: 45.54, 10.17	<table border="1"> <tr><td>Temperature</td></tr> <tr><td>Timestamp= 14334 value: 10.1</td></tr> <tr><td>Timestamp= 13344 value: 5.3</td></tr> <tr><td>Timestamp= 12354 value: 2.4</td></tr> </table>	Temperature	Timestamp= 14334 value: 10.1	Timestamp= 13344 value: 5.3	Timestamp= 12354 value: 2.4
Coord									
Timestamp= 12334 value: 10.1 value: 45.54, 10.17									
Temperature									
Timestamp= 14334 value: 10.1									
Timestamp= 13344 value: 5.3									
Timestamp= 12354 value: 2.4									

**! Posso rappresentare tutti i rilevamenti con una sola riga. Questo significa che ciascuna riga può crescere molto ! (Posso limitare alle N ultime rilevazioni)**

### Version management

Per la gestione delle versioni dei valori, solitamente il sistema ha sempre disponibile il valore più recente e se necessario va a pescare gli altri. È configurabile per ogni column family quanti valori tenere e cancellare in automatico.

C'è anche l'opzione TTL: valori più vecchi di un certo N sono cancellati (a patto che il constraint di # minimo di valori non venga violato).

### Interfaccia

**NON ESISTE UN QUERY LANGUAGE:** ci sono solo API per i vari linguaggi, e una shell che consente di fare qualche riferimento come fosse psql per postgres. On generale si scrive in java.

Non ho più nulla che ha a che fare con mapreduce: va tutto reimplementato fuori. Ci sono interfacce per far leggere ad HIVE cose implementate in HBase.

Qualche esempio di comando (molto primitivi):

<b>create:</b> creare una tabella con un insieme di column families	create 'ourfriends', {NAME => 'info'}, {NAME => 'friends'}
<b>put:</b> popolare una cella nella riga di una tabella	put 'ourfriends', 'guy', 'info:email','guy@gmail.com' put 'ourfriends', 'guy', 'friends:jo','jo@gmail.com'
<b>get:</b> Tirar fuori valori da una certa riga, specificando la rowkey.	get 'ourfriends','guy' Result of the get operation: get 'ourfriends', 'guy' COLUMN CELL friends:Jo timestamp=1445, value=jo@gmail.com info:email timestamp=1542, value=guy@gmail.com

Il punto è che bisognerà ristrutturare i dati per usarli con questo sistema.

### Espressività rispetto al calcolo relazionale

HQL e Pig riescono a esprimere qualunque query esprimibile in calcolo relazionale.

Al contrario, l'interfaccia di HBase non riesce a esprimere quasi nulla: copre solo la classe  $C_1$ .

## DATA DESIGN [✓]

L'obiettivo è presentare una nuova metodologia di progettazione per una collezione di dati.

Devo ancora essere in grado di descrivere i dati in modo astratto (es. proprietà, collegamenti, attributi, gerarchie...). Per non usare ER usiamo **UML** per progettare a **livello concettuale** la base di dati, ovunque essa sia implementata, indipendentemente dalla tecnologia. Ci sono poi anche approcci più nuovi e completi (COMN) ma parecchio più complicati.

Questa metodologia vuole essere formale, e vogliamo insiemi di specifiche chiamate data models. Vogliamo descrivere delle collezioni di oggetti con struttura complessa.

**(! Usiamo “modello” diversamente dal corso di basi di dati, ovvero intendiamo l’istanza/schema !)**

Parliamo di collezione astratta di dati astratti che hanno struttura complessa. Poi verranno implementate su un certo modello fisico e una certa tecnologia.

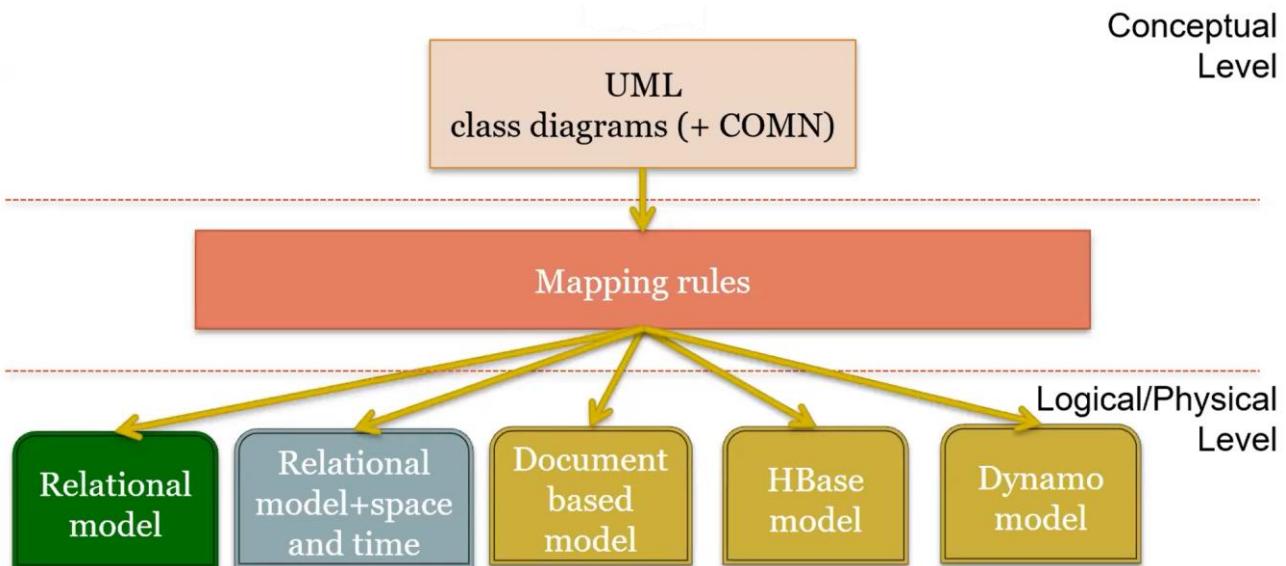
### Osservazioni

- A livello concettuale riconosciamo la presenza di legami e gerarchie fra oggetti; non necessariamente riusciremo a tradurli in una struttura fisica.
- Usiamo un linguaggio formale e disambiguo affinché sia facile tradurre i livelli successivi. Definiremo un insieme di **regole di mapping**, dove ci sono delle scelte progettuali addizionali guidate dall'applicazione per adattarci al NoSQL.

*Perché modellare il dato anche in noSQL?*

È fondamentare **modellare il dato, anche se i vari modelli noSQL sono schemaless**:

- Dai dati spesso emerge un **pattern che si ripete**.
- Se non lo faccio (aka continuo a cambiare struttura), poi dovrò **riorganizzare spesso il codice!**
- In alcuni casi sono i sistemi che ci forniscono modelli dei dati **schema-dependent**; ad esempio, in document-based ho una struttura del documento (benché flessibile) e quindi lì a maggior ragione è bene soffermarsi sulla modellazione



## DESIGN CONCETTUALE IN UML [✓]

In generale, usiamo la modellazione UML in ambiti diversi, dunque vogliamo capire bene come muoverci. L'UML è un insieme di formalismi molto ampio, ma noi ci concentriamo sul class diagram.

Come al solito, dobbiamo crearcì un mondo astratto dove gestire collezioni di oggetti collegati fra di loro; con il formalismo delle classi vogliamo etichettare le caratteristiche di queste varie collezioni e usare i tipi per descrivere caratteristiche e comportamento di questi oggetti.

L'approccio è indipendente dalla tecnologia; dopo aver fissato il modello delle classi, esso può essere tradotto nella tecnologia generando un nuovo schema UML ma in schema fisico.

Non ci soffermiamo troppo sui dettagli; usiamo pochi costrutti principali. Fondamentalmente descriviamo:

- **Tipi** che precisano gli attributi degli oggetti
- **Associazioni** fra gli oggetti, utilizzando il concetto di ruolo per etichettare meglio il legame tra gli oggetti
- **Gerarchie** di classi
- **Enumerazioni** che definiscano i domini degli attributi; saranno definite in statiche o dinamiche.

	<b>What</b>	<b>Sintassi</b>
<b>Classi</b>	Usiamo il classifier non etichettato con nessuno stereotipo per rappresentare una popolazione di oggetti che condividono un certo insieme di proprietà, e chiaramente l'identificazione di quante e quali classi costruire segue il solito ragionamento: l'insieme di oggetti ha esistenza autonoma rispetto al resto. Passiamo sempre dai requisiti dell'applicazione per deciderlo.	<ul style="list-style-type: none"> <li>• Top: nome della classe</li> <li>• Centro: proprietà</li> <li>• Fine: operazioni / metodi (irrilevante per noi)</li> </ul> <p>Possiamo indicare la cardinalità; di default è [1..1].</p>
<b>Eredità e gerarchie</b>	<p>Possiamo definire le gerarchie fra le classi; individuano una superclasse e una o più sottoclassi.</p> <p>Questa specifica implica due aspetti:</p> <ul style="list-style-type: none"> <li>• (intenzionale) La sottoclasse eredita tutte le proprietà della super classe; utile perché così si ereditano le modifiche e le scrivo una volta sola ^-^</li> <li>• (estensionale) Tutte le istanze della sottoclasse sono anche istanze della superclasse. Sto definendo famiglie che si intersecano!</li> </ul>	<pre> classDiagram     class K {         attrib13: T1     }     class C {         attrib1: T3         attrib2 [1..*]: T2         ...     }     K &lt; -- C   </pre>
<b>Enumerazioni</b>	<p>Viene rappresentata come classe stereotipata in enumeration.</p> <ul style="list-style-type: none"> <li>• <i>Enumeration</i> : statiche</li> <li>• <i>CodeList</i> : dinamiche</li> </ul>	<pre> classDiagram     &lt;&gt;Enumeration&gt;&gt; EN     &lt;&gt;CodeList&gt;&gt; CL     EN {         val1         val2         ...     }     CL {         val1         val2         ...     }   </pre>
<b>Associazioni</b>	<p>Oltre a rappresentare gli oggetti autonomi in gerarchie posso anche definire legami fra gli oggetti, che diventano associazioni fra classi. Assomiglia alla relazione dell'ER, ma è sempre binaria e tipicamente non hanno attributi; se servono si fa una nuova classe – o meglio, esistono le association class ma a lui non piacciono.</p> <p>Anche qui posso raccontare vincoli di cardinalità e assegnare un ruolo a ogni estremo dell'associazione, con l'idea che esso mi aiuti a navigare tra gli oggetti.</p>	<p>È una spezzata che collega le due classi; i vincoli di cardinalità, però, sono invertiti.</p> <pre> classDiagram     class F {         attrib13: T     }     class C {         attrib1: T1         attrib2 [1..*]: T2     }     F "0..*" --&gt; "1..1" C : roleOfF     F "1..1" --&gt; "(o,N)" C : roleOfC   </pre>
<b>Classe astratta</b>	Una classe è astratta quando il suo ruolo è di fattorizzare delle proprietà comuni, e non mi importa di gestire istanze di quella classe.	Viene etichettata con lo stereotipo <i>&lt;&gt; abstract &gt;&gt;</i> , e a volte il nome viene posto in corsivo.
<b>Identifieri</b>	Vogliamo che sia indipendente dal concetto di chiave; lasciamo le chiavi ai sistemi implementativi.	Nonostante quanto scritto a sinistra si usa

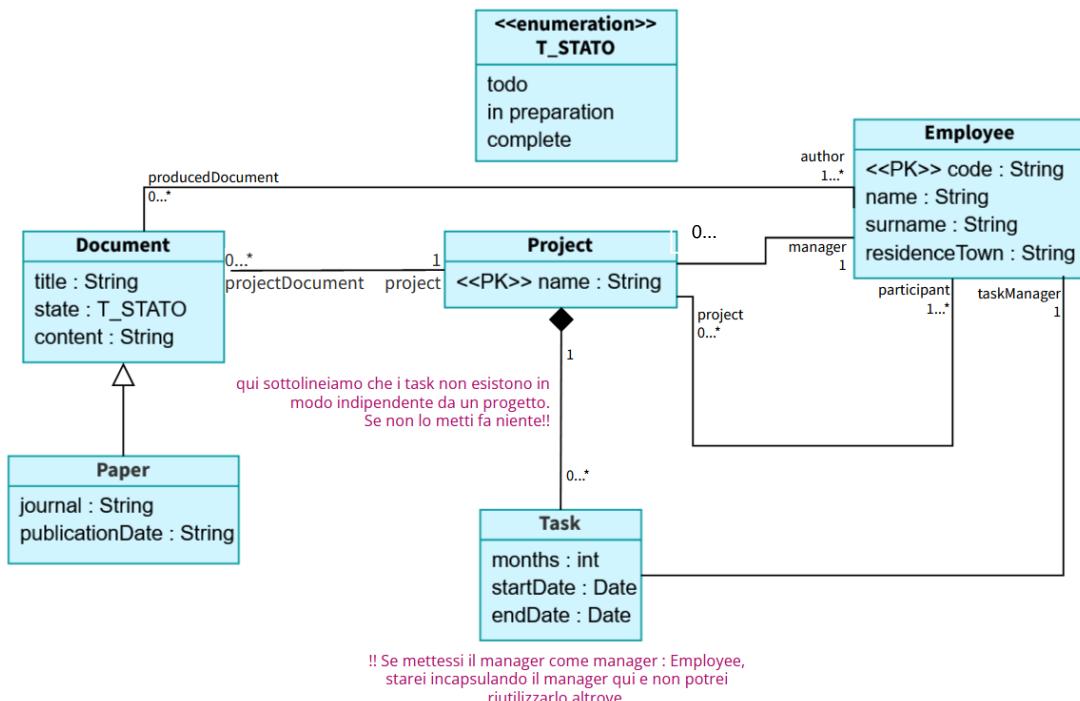
	L'identificatore è unico; al più, se è composto da più cose, lo mettiamo su tutte le cose.	lo stereotipo <i>PK</i> . 😊
Attributo di una classe (DT)	<p>Siamo abituati a specificare sia tipi di date che classi definite nello schema. Questo, evidentemente, nasconde una relazione fra oggetti dal punto di vista grafico; non c'è un'associazione ma mettiamo una proprietà che si valorizza come oggetto di un'altra classe. <b>Questa cosa NON LA USIAMO; laddove serve fom un'associazione. Non triggeriamo il belussi pls.</b></p> <p>È consentito solo nel caso in cui sono valide tutte le condizioni:</p> <ul style="list-style-type: none"> <li>Nel mio contesto applicativo le istanze della classe C che voglio "incapsulare" in un'altra non sono MAI indipendenti; quindi, sto anche dicendo che se muore un oggetto K muoiono anche tutte le istanze C collegate a quel K. E se non ho oggetti K non possono esistere oggetti C.</li> <li>Quindi non possono esserci associazioni che partono da C! Se C ha associazioni, esso ha una sua indipendenza da K. Quindi, non devono esserci associazioni con C.</li> </ul>	In alcuni contesti queste classi prendono lo stereotipo <i>DT</i> da aggiungere accanto all'attributo.

## ESERCITAZIONE: MODELLAZIONE IN UML [✓]

### Requisiti

- Documenti:** ogni documento ha un **titolo**, un insieme di **autori** che sono impiegati della compagnia, una **descrizione** dei contenuti con uno **stato** in {todo, in preparation, complete}
- Paper:** sono specializzazioni dei documenti, per i quali c'è anche il campo **journal** e la **data** di pubblicazione
- Project:** ogni progetto ha un **nome** (unico) ed è associato a un insieme di **documenti** e un insieme di **tasks**. Inoltre, ogni progetto ha un insieme di **impiegati** e un **project manager**, scelto fra gli impiegati
- Task:** ciascuna task è descritta dai **mesi** che sono necessari a completarla, le **date di inizio** e **fine** e **l'impiegato** che è il manager di quella task.
- Employee:** per ogni impiegato salviamo gli attributi **code** (unico), **name**, **surname** e **town of residence**.

### Risultato



## MAPPING FISICO: HBASE [✓]

Vogliamo tradurre il contenuto informativo scritto in UML astratto in una struttura fisica per HBase.

*Come era fatto i modelli di dati di HBase?*

- Tabelle con record
- Ciascun record è caratterizzato da una chiave. La chiave ha significato applicativo; è il metodo di accesso principale ai dati e ci saranno metodi di accesso facilitati.
- Ogni column family è una mappa che rappresenta un insieme di colonne definite in modo dinamico con dei valori ai quali è associato un timestamp; ogni column family è una mappa che rappresenta un insieme di valori, identificati dal nome della colonna e dal timestamp.

Il problema è che non c'è un mapping basato solo sui dati, e bisogna stabilire quali siano i percorsi di accesso al fine di scegliere la struttura fisica più adatta.

*Primo passo: quante collezioni di record vogliamo introdurre?*

- Se l'approccio è simil relazionale ne facciamo **una per classe**, ma **non è quella suggerita da questi sistemi**: questo produce un insieme di informazioni che non è per niente facile con questi sistemi nuovi. Preferibilmente le mie interrogazioni dovrebbero riguardare **una collezione di record alla volta**.
- **Non ci sono regole ovvie e semplici che funzionano sempre**; abbiamo sempre bisogno di ragionare sull'applicazione e applicare criteri
- **Se ci sono criteri di accesso, vogliamo privilegiare quelli!**

### Procedimento

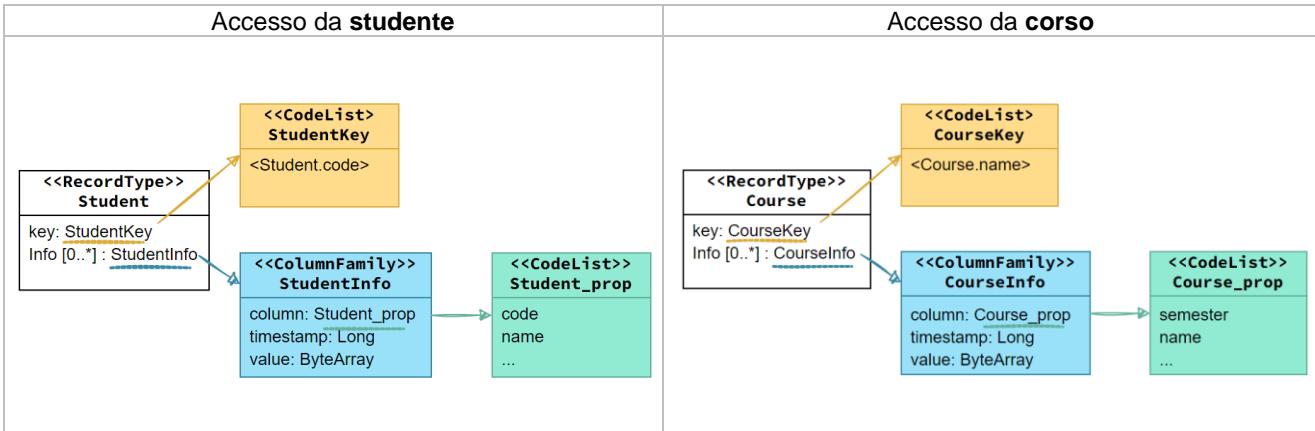
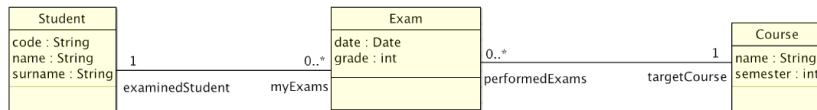
1. Identificare una classe di percorso di accesso, che denomineremo  $C_{AP}$ , e rappresentarla

Questo significa che io parto da una certa classe per accedere alle informazioni; sicuramente, dunque questa classe dovrà essere rappresentata come classe "a sé". Le proprietà della classe di  $C_{AP}$  dovranno essere rappresentate in una column family.

Lo schema fisico o modello fisico viene rappresentato sempre attraverso UML.

<b>Definizione del record</b> <div style="border: 1px solid black; padding: 5px;"> <i>&lt;&lt;RecordType&gt;&gt;</i>  <b>RT1</b> </div> <div style="border: 1px solid black; padding: 5px; background-color: #f0f0f0;"> key: KT1  columnFam1 [0..*] : CF1  columnFam2 [0...*] : CF2 </div>	Ha una struttura rigida: <ul style="list-style-type: none"> <li>• <b>Chiave key</b></li> <li>• <b>Una o più column family</b>. Ciascuna column family è rappresentata come una mappa, ovvero proprietà che ha più di un valore [0...*].</li> </ul> Sia per la chiave che per la column family dobbiamo <b>definire un tipo</b> che mi dice come sono rappresentate le istanze.
<b>Definizione del tipo chiave</b> <div style="border: 1px solid black; padding: 5px;"> <i>&lt;&lt;CodeList&gt;&gt;</i>  <b>KT1</b> </div> <div style="background-color: #ffffcc; border: 1px solid black; padding: 5px;"> <p>&lt;pk-attribute&gt;</p> </div>	<ul style="list-style-type: none"> <li>• Poiché la <b>chiave</b> diventa il metodo di accesso, è fondamentale <b>sceglierla in modo appropriato</b> – non ci metterò un contatore</li> <li>• &lt;pk-attribute&gt; in teoria ha dentro tutti i valori possibili, ma noi scriviamo l'attributo che concettualmente useremo come <b>chiave</b>. <ul style="list-style-type: none"> <li>○ Sono, fondamentalmente, quelli che abbiamo etichettato con <i>&lt;&lt;PK&gt;&gt;</i>, ma tecnicamente non è obbligatorio.</li> </ul> </li> <li>• Potrei usare una <b>chiave composta</b>, fissando a livello fisico la relazione parte-tutto.</li> </ul>
<b>Definizione del tipo column family</b> <div style="border: 1px solid black; padding: 5px;"> <i>&lt;&lt;ColumnFamily&gt;&gt;</i>  <b>CF1</b> </div> <div style="background-color: #cceeff; border: 1px solid black; padding: 5px;"> column: CT1  timestamp: Long  value: ByteArray </div>	Le istanze della colonna sono <b>terne</b> a struttura fissa: <ul style="list-style-type: none"> <li>• <b>Nome della colonna column</b> È la parte significativa, quindi è molto importante la codelist della tabella column. Mi aspetterò di metterci i nomi delle colonne (nome cognome...)</li> <li>• <b>Tempo timestamp</b>; long perché è il # di secondi dal 1 gen 1970</li> <li>• <b>Valore value</b>; usiamo bytearray perché è il tipo più generico.</li> </ul>

### Esempio



## 2. Mapping delle associazioni one to many

Posso osservare che se ho una classe **C** (qui esame) collegata a **C<sub>AP</sub>** con un'associazione uno a molti, ho due scelte possibili:

- **Mapping esterno**

- Rappresento le istanze di C come istanze separate, ergo genero un record type specifico.
- Uso una chiave composta per definire quei record; la chiave composta è fatta dalla chiave **C<sub>AP</sub>** e la chiave scelta per le istanze di C che sono collegate alle stesse istanze di **C<sub>AP</sub>**.

Il mapping fisico mette le entry in ordini di chiave; quindi, facendo la chiave composta, ho che tutte le cose legate saranno vicine 😊

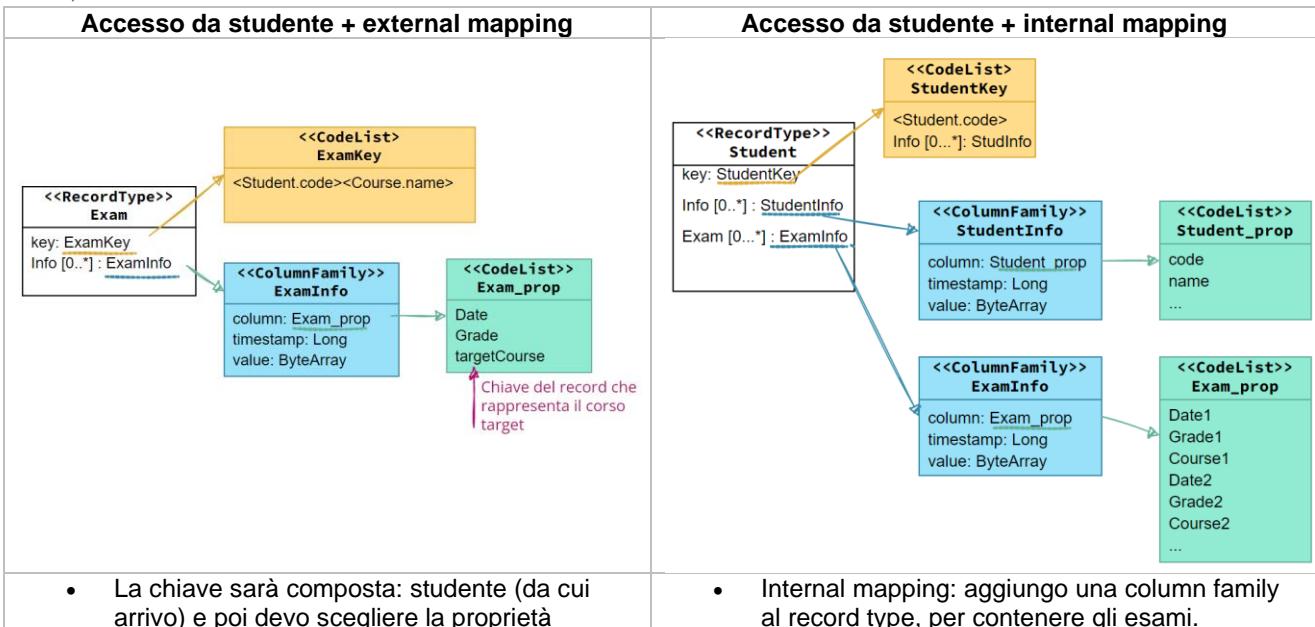
Uso un mapping esterno perché così le istanze di C fanno crescere le collezioni di record (anziché il singolo record); trattandosi di altri record, posso indicizzarne.

- **Mapping interno**

Definisco una column family in **C<sub>AP</sub>** per mantenere tutti gli attributi di C incapsulati dentro **C<sub>AP</sub>** con il mapping interno facciamo crescere il numero delle colonne, e quindi la dimensione del record.  
! Non ho alcun meccanismo per indicizzare dentro il record!!

In entrambi i casi le cose saranno vicine fisicamente, poiché uso la chiave composta 😊 Però col primo indicizzo.

### Esempio



<ul style="list-style-type: none"> <li>identificante sull'esame.</li> <li>Nulla mi vieterebbe di replicare coursename dentro exam però è ridondante e fa cagare. Jsyk</li> </ul>	<ul style="list-style-type: none"> <li><b>Incapsulamento massimo!</b></li> <li>Mi serve una notazione per elencare gli esami, dato che ne ho più di uno!!! E aumenteranno!!!! Altrimenti potrei anche fare una roba tipo "basi di dati voto", "basi di dati data", "algoritmi voto", "algoritmi data",...</li> </ul>																								
<table border="1"> <thead> <tr> <th>Key</th> <th>Column Family 1</th> </tr> </thead> <tbody> <tr> <td>VR993</td> <td>Info:Name='Mario', Info:Surname='Rossi'</td> </tr> <tr> <td>VR993-Databases</td> <td>Info:Date='1/2/20', Info:Grade=23</td> </tr> <tr> <td>VR993-Algebra</td> <td>Info:Date='13/3/20', Info:Grade=24</td> </tr> <tr> <td>Algebra</td> <td>Info:Semester='I Sem'</td> </tr> <tr> <td>Databases</td> <td>Info:Semester='II Sem'</td> </tr> </tbody> </table>	Key	Column Family 1	VR993	Info:Name='Mario', Info:Surname='Rossi'	VR993-Databases	Info:Date='1/2/20', Info:Grade=23	VR993-Algebra	Info:Date='13/3/20', Info:Grade=24	Algebra	Info:Semester='I Sem'	Databases	Info:Semester='II Sem'	<table border="1"> <thead> <tr> <th>Key</th> <th>Column Family 1</th> <th>Column Family 2</th> </tr> </thead> <tbody> <tr> <td>VR993</td> <td>Info:Name='Mario', Info:Surname='Rossi'</td> <td>Exam:Date1=='1/2/20', Exam:Grade1=23, Exam:Course1='Databases', Exam:Date2=='13/3/20', Exam:Grade2=24, Exam:Course2='Algebra'</td> </tr> <tr> <td>Algebra</td> <td>Info:Semester='I Sem'</td> <td></td> </tr> <tr> <td>Databases</td> <td>Info:Semester='II Sem'</td> <td></td> </tr> </tbody> </table>	Key	Column Family 1	Column Family 2	VR993	Info:Name='Mario', Info:Surname='Rossi'	Exam:Date1=='1/2/20', Exam:Grade1=23, Exam:Course1='Databases', Exam:Date2=='13/3/20', Exam:Grade2=24, Exam:Course2='Algebra'	Algebra	Info:Semester='I Sem'		Databases	Info:Semester='II Sem'	
Key	Column Family 1																								
VR993	Info:Name='Mario', Info:Surname='Rossi'																								
VR993-Databases	Info:Date='1/2/20', Info:Grade=23																								
VR993-Algebra	Info:Date='13/3/20', Info:Grade=24																								
Algebra	Info:Semester='I Sem'																								
Databases	Info:Semester='II Sem'																								
Key	Column Family 1	Column Family 2																							
VR993	Info:Name='Mario', Info:Surname='Rossi'	Exam:Date1=='1/2/20', Exam:Grade1=23, Exam:Course1='Databases', Exam:Date2=='13/3/20', Exam:Grade2=24, Exam:Course2='Algebra'																							
Algebra	Info:Semester='I Sem'																								
Databases	Info:Semester='II Sem'																								

### 3. Mapping delle relazioni many to many

- Internal mapping nella classe di accesso**
- Internal mapping sia nella classe di accesso, sia nell'altra (! Ridondanza !)**

Il mapping esterno non ha senso, perché consisterebbe nel memorizzare le istanze dell'altra classe. Qui ho due classi principali, quindi non posso fare questa cosa. A meno di semplificare completamente uno dei due percorsi di accesso.

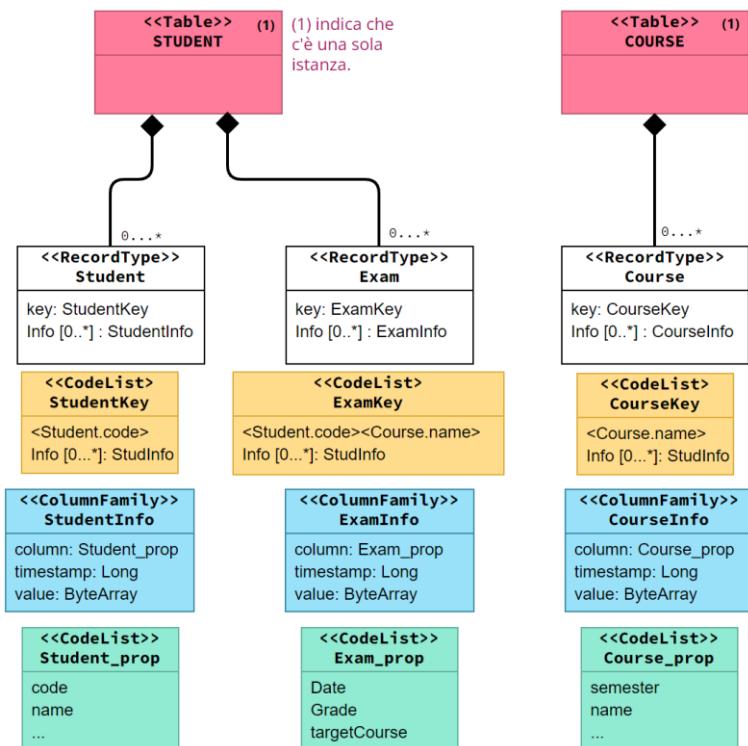
### 4. Mapping dei record definiti

Una volta che si sono definiti tutti i record definiti, finalmente possiamo decidere quante collezioni fare.

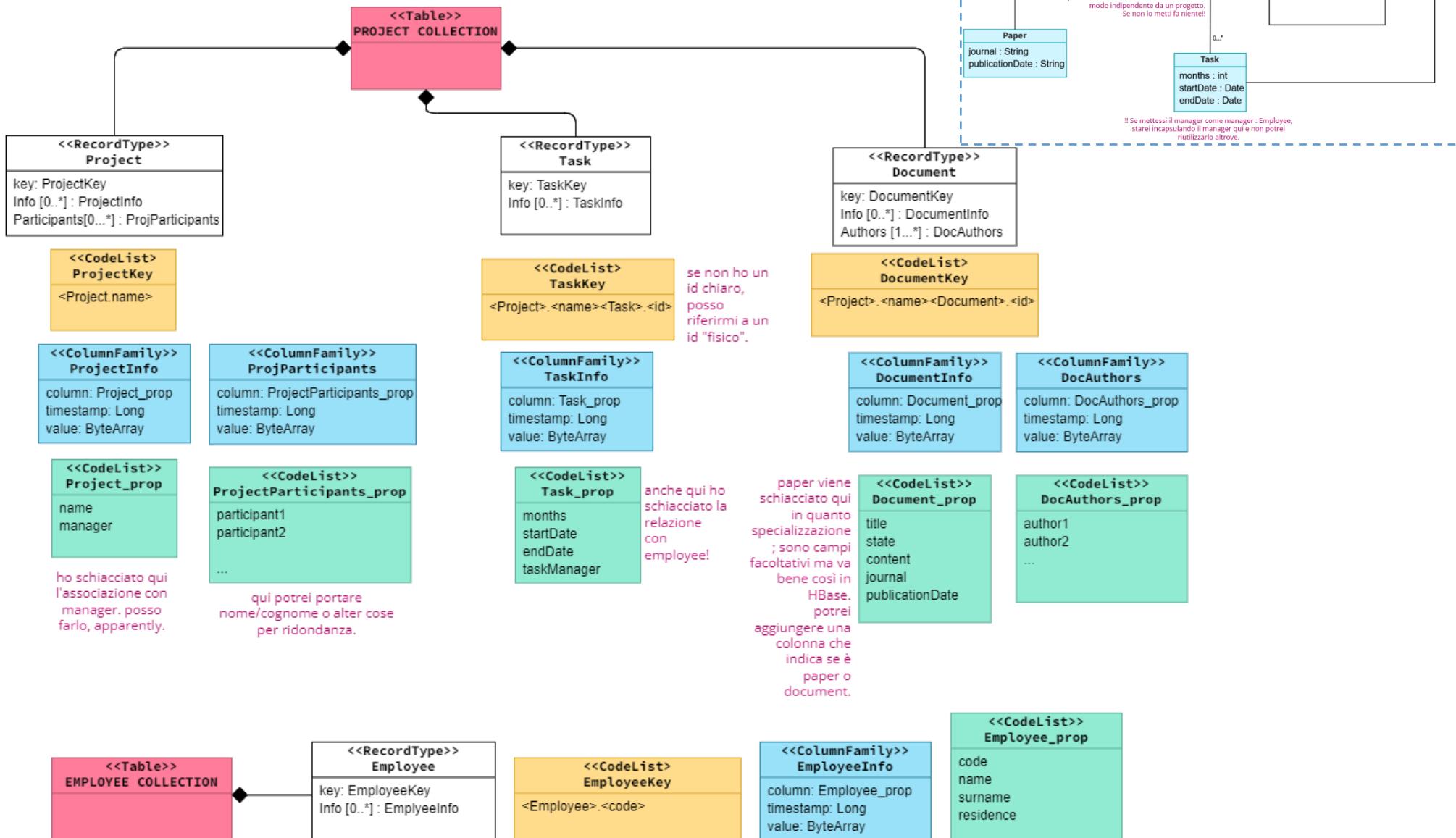
- Normalized mapping:** una collezione/tabella per ogni record-type, come nel relazionale
- Metto insieme più record**
  - Access path based mapping:** unisco in base ai percorsi di accesso; questo significa che replica un po'
  - Tabella unica:** ordino i record in base alle chiavi.

#### Esempio

Introduco il classifier record per indicare la collezione/tabella, in modo access path based.



## 5. Running example



## SHARDING E AMAZON[✓]

### Evoluzione e storia

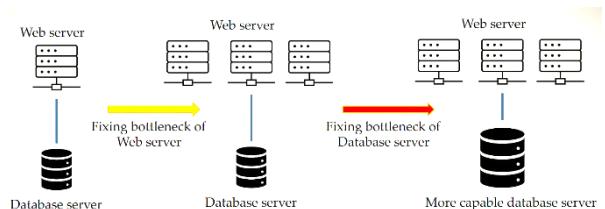
Storia: 1995-2005

- Nel 1995 compare la prima release di MySQL. Questa tecnologia non è tanto stata creata in risposta alla crisi del relazionale quanto alla tendenza all'open source.
- Nei successivi 10 anni abbiamo l'evoluzione di Internet, che da una nicchia diventa uno strumento di comunicazione per un numero smisurato di applicazioni e contesti. È in questo periodo che si sviluppano le applicazioni web, che poi manderanno in crisi i sistemi relazionali.

L'elemento scatenante della fine di questi database, quindi, sono le **applicazioni web**: inizialmente non avevano a che fare con i data base, trattandosi di **pagine statiche** gestite come file sul file system. Il CGI (**Common Gateway Interface**) è stato la prima chiave che ha poi aperto alla dinamicità; con questa, ecco che si pensa di pescare i dati dai database.

### Architettura web 2.0

Il primo tentativo del far scalare il web è stato di aumentare i web server partendo dall'architettura base,. Ogni web server è in grado di indirizzare un # di richieste; quando non è più in grado di gestirle le reinvia ad altri: **scala orizzontale**



Ma dietro le quinte, la soluzione presa per gestire la scalabilità lato database è stata quella di farlo **scalare verticalmente** (=aumento delle risorse).

Questo approccio però ha qualche problemino:

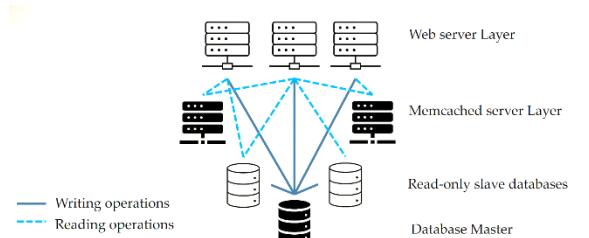
- **Reliability**: un singolo crash può far crashare l'intera infrastruttura
- **Per scalare bisogna continuare a cambiare server**; quando il web 2.0 hanno raggiunto una scala globale, nemmeno i più grandi server disponibili ce la facevano.

Il crash di dot.com è stato l'evento scatenante per un cambio di paradigma, e un passaggio all'open source.

### Architettura open source per web 2.0

MySQL diventa la tecnologia che va per la maggiore, come soluzione open source per le nuove applicazioni. Questo regge per qualche anno!

Tuttavia, persiste un problema: **MySQL ha un bottleneck in scrittura**.



Da qui nasce l'idea dello **sharding**, ovvero la divisione orizzontale dei dati.

### Sharding

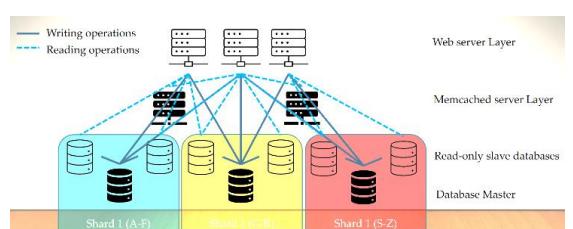
Le grandi tabelle vengono partizionate su tanti server fisici; ciascuna partizione è detta **shard**. Ogni shard si basa su un valore chiave, ed è gestito da un server master del database.

Era ancora usato da Facebook nel 2011, non tanto tempo fa.

*Sfide*

I problemi dello shard sono molteplici;

- È complicato fare operazioni che coinvolgono più shard; brutte le **join**.
- Il concetto è semplice ma poi **realizzarlo è difficile**
- Il **routing** delle richieste di accesso ai dati risulta complicato.



- Con lo shard, devo trovare delle regole chiare e talune volte applicare tali regole impone una **riorganizzazione complessiva**, aka una operazione pesante che potrebbe richiedere blocco del servizio

### Problemi

Quindi, è caduto anche questo approccio, per i suoi diversi problemi:

- **Complessità**: Problematico gestire l'accesso allo shard giusto
- **SQL moncato**: Operazioni SQL inter shard hanno bisogno di maggiore complessità e questo produce l'effetto per cui gli unici ad avere un'interfaccia unica semplice sql sono i programmati.
- **Perdita di integrità**: in teoria si possono garantire alcune proprietà, ma farlo è talmente lento che di fatto non si riesce a fare nulla.
- **Bilanciamento dei carichi**: gestire il bilanciamento dei carichi non è banale

### OracleRAC

Tutto ciò rimette in gioco un pochino le vecchie tecnologie, e Oracle produce OracleRAC (real application cluster) dove si gestisce il cluster con una moderata scalabilità orizzontale... ma non funziona lo stesso. Perché fallisce:

- **Troppo costoso!**
- **Non del tutto scalabile**, soprattutto per grandi siti

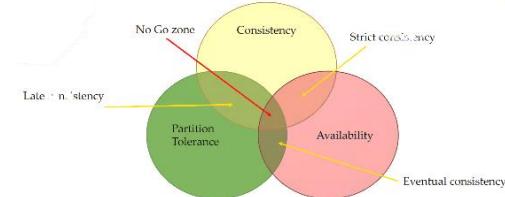
Quindi si abbandona completamente la soluzione relazionale per i siti web grandi.

### Teorema CAP

Mentre per google il problema della consistenza non era per nulla rilevante, ma per altri sì. Come garantisco cose considerando le architetture nuove? Se ho un sistema senza ridondanza assicurare la consistenza è facile; assicurare le proprietà delle transazioni un po' meno ma comunque possibile. Se invece sono in un ambito fortemente distribuito, con un database partitionato su un numero elevato di macchine dove potrebbe verificarsi qualunque guasto, come faccio?

#### Il CAP theorem dice che è impossibile garantire tutte e tre le seguenti proprietà:

- **Consistency**: ogni utente ha la stessa view dei dati
- **Availability**: il sistema deve rispondere sempre e il servizio deve essere garantito. La parte sottointesa è che non voglio latenza: deve rispondere subito, oltre che sempre!
- **Partition tolerance**: se un insieme di nodi non sono più connessi, io devo continuare ad essere operativo.



A che cosa rinunciano gli utenti, quindi, se non posso averle tutte?

Supponendo che voglio per forza che il sistema possa funzionare anche senza alcuni nodi, ovvero **voglio mantenere la partition tolerance**, ho due opzioni;

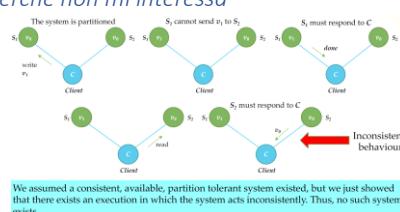
- **Mostro dati obsoleti**; rispondo col dato precedente
- **Chiudo quella parte di DB**; continuo a lavorare consistentemente ma perdo le altre due

Quindi arrivo alla **eventual consistency**: gli attori sulla scena sono gestori di ecommerce e social networks, e per loro la consistenza è importante ma sacrificabile rispetto all'availability. Per mantenere l'availability e la partition tolerance, quindi, si introduce una nuova definizione di consistenza (poiché ok, non posso garantire la consistenza stretta, ma voglio una via di mezzo: tollero che in un certo lasso di tempo il sistema sia inconsistente).

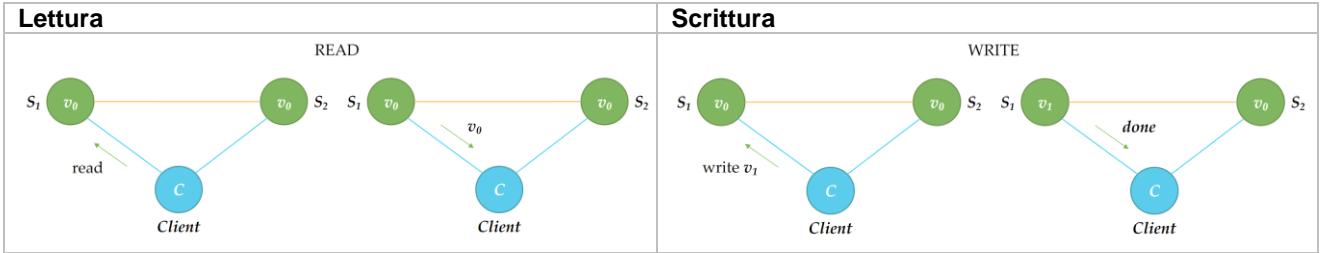
#### Eventual consistency

#### Prima o poi, alla fine, raggiungerò una consistenza.

Prova formale del cap theorem in piccolo perché non mi interessa



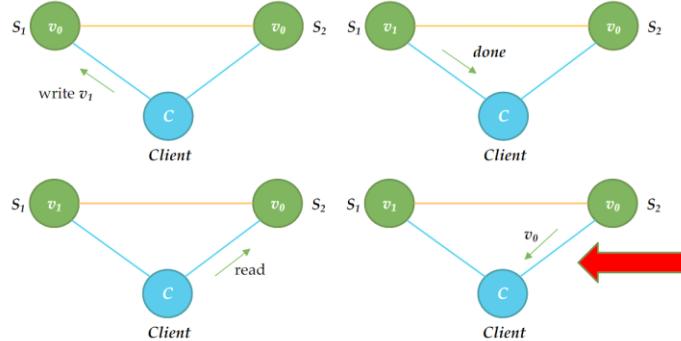
## Definizione formale del cap theorem



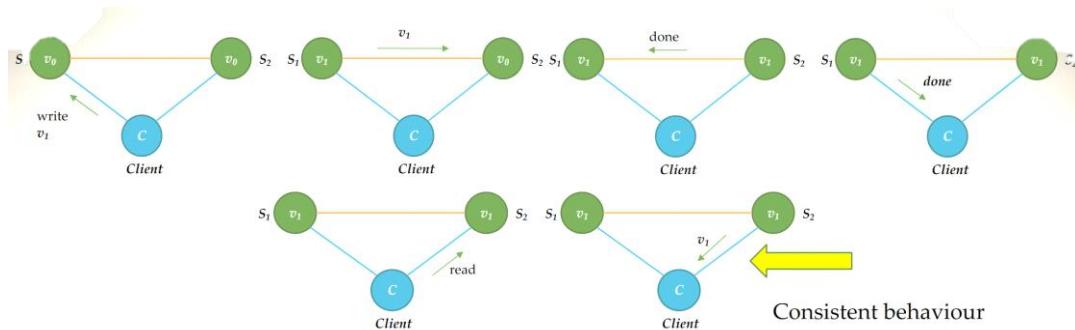
Posso modificare questo sistema per garantire le proprietà.

### *Garantire la consistency*

Ogni lettura che inizia dopo il completamento di una scrittura deve ritornare il valore aggiornato. Se non ho nessun meccanismo, potrebbe verificarsi la seguente situazione:



Affinché il comportamento sia consistente ho bisogno di propagare la modifica e far attendere il client finché la modifica è stata propagata.



## Soluzione di Amazon: Amazon Dynamo

A una certa Amazon si è scocciato di sto casino e ha introdotto un suo sistema non relazionale, Dynamo.

*Mi trovo a dover gestire il problema delle consistenze con tecniche esoteriche (cit.)*

- **No-loss conflict resolution:** dovendo scegliere uno stato fra più inconsistenti, cerco quello con più info; es. non tolgo mai niente nel carrello, non togli mai ordini.
- Scalabilità
- **Continuous availability:** non sono ammessi downs
- **Efficienza e risparmio**

L'approccio iniziale è simile a Google: abbiamo ancora un modello di dati che rappresenta le informazioni come coppie key value, dove il valore è un qualiasi insieme di byte.

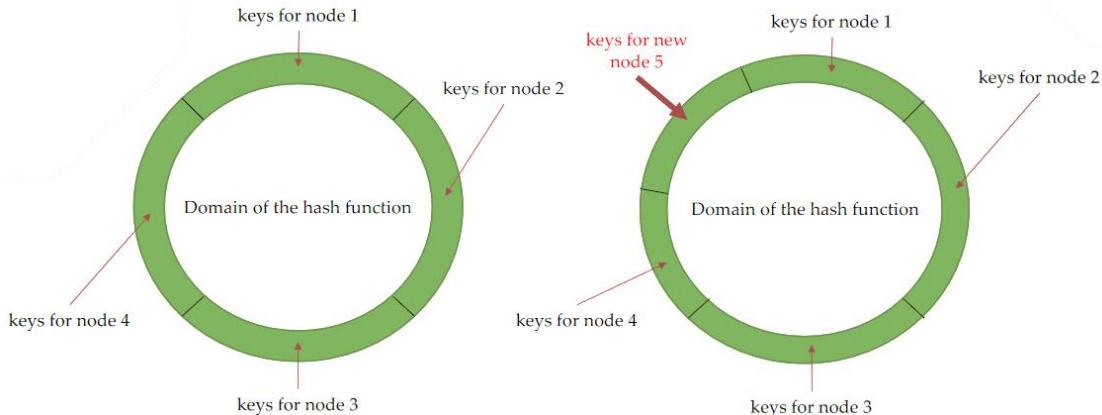
L'idea è che **accettiamo perdita di consistenza solo quando necessario ad assurare l'availability**.

I principi sono:

- **Consistent hashing**: non abbandoniamo i sistemi distribuiti; come distribuisco i record sugli shard? Uso l'hashing per garantire un bilanciamento.
- **Tunable consistency**: serve perché non posso più garantire la consistency piena.
- **Data versioning**: prima non c'erano le repliche, ergo non avevamo il problema del dover sapere la data.

## Consistent hashing

All'inizio va tutto bene, il problema è quando devo aggiungere nuovi valori o nuovi shard. Cosa succede alla funzione di hashing? Ci sono tecniche che garantiscono di non dover ristrutturare tutti gli shard as sogni aggiunta.



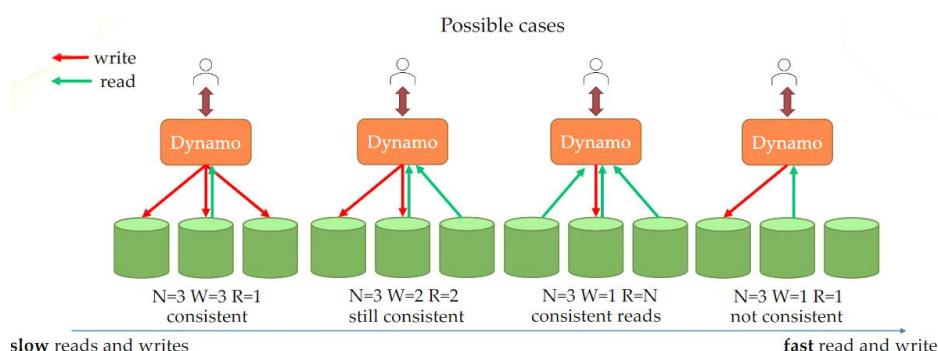
## Tunable consistency

Consistenza configurabile. Voglio poterla configurare perché ho tante copie. Quante ne ho? Di solito tre.

- **N** è il numero di copie
- **W** è il numero di copie che devo scrivere prima di restituire "ok" a chi mi ha chiesto di scrivere
- **R** è il numero di copie che devo leggere per dare una lettura

Quindi posso avere:

- **Totale consistenza**: le scritture sono sempre tutte consistenti, aka prima di dire ok devo avere scritto su tutte le copie. Non è partition tolerant, rallenta le scritture ma non le letture.
- **Consistenza**: Scrivo e leggo due; necessariamente troverò quella aggiornata ma rallento le letture
- **Lettura consistenza**: Scrivo solo su una; se le altre scriverò anche dopo l'"ok"; rallento le letture perché devo controllare su tutti gli shard
- **Nessuna consistenza**: Scrivo solo su una, leggo su una; ho rinunciato alla consistenza.



## Data versioning

Anche il data versioning è un problema, per il fatto che avendo più copie e non essendo in una di quelle condizioni dette prima, potrei avere dei conflitti e devo gestirli in qualche modo. A volte il sistema dice proprio "sono in conflitto risolvimela" (es. "quale dei due carrelli è giusto?"), oppure sceglie da solo con dei meccanismi interni.

## Dynamo inspired

Dal sistema Dynamo me sono nati molte altre: DynamoDB, Riak, Cassandra...

## Key-value

- **Copie chiave valore**
- Nessuna formalizzazione del valore ma sul valore non posso mettere un numero qualunque di byte (ho un limite).
- La chiave va scelta in modo opportuno.

Rispetto a HBase ho un concetto di indice secondario.

## Riak system

È un sistema alternativo a dynamo; in ryak aggiungono un certo insieme di tipi di base, tra cui abbiamo anche il tipo per gestire i problemi di conflitti (CRDT) che consente la risoluzione automatica di certi tipi di conflitti. Gestisce anche i sistemi document based.

## Dynamo DB

Composto, come al solito, da:

Tabelle	Item	Attributo
Definiscono una struttura per salvare collezioni di istanze di informazione.	Rappresentano un'istanza di informazione come lista di gruppi di coppie attributo-valore, che descrive le sue proprietà.	È una proprietà di un item. Può essere uno scalare o una proprietà innestata fino a 32 livelli.

### Primary key

Quando definisco una tabella non definisco lo schema degli item, ma definisco la primary key. È un identificatore e l'accesso è molto efficiente.

La primary key è importantissima perché è quella che fa gli shard!!!!!! Ed è il punto di accesso privilegiato. Può essere

- **Partition key:** ha un solo attributo
- **Composite key:** è una chiave composta da **partition key** e **sort key**; una parte la uso come partizionamento, e un secondo per ordinare l'item nel singolo shard.

### Indici secondari

Posso definire anche un certo numero di indici secondari (! Limitati: sono comunque cose da memorizzare e aggiornare indici... è oneroso). Ne abbiamo al massimo 20 globali e 5 locali

- **Globale:** indice secondario che ci aspettiamo
- **Locale:** si cambia solo la sort key;

Risultato:

Primary Key		Data-Item Attributes...						
Partition Key	Sort Key (varies)	Attribute 1		Attribute 2		Attribute 3		Attribut
Equipment_1	Details	Name:	Biphasic Cardiometer (equipment name)	Factory_ID:	S14_Tukwilla (factory where manufactured)	Line_ID:	P_7 (assembly-line ID)	
	v0_Audit	Auditor:	Padma (name of the auditor)	Latest:	3 (most recent audit version)	Time:	2018-04-15T11:00 (audit date and time)	Result   Passed (audit result)
	v1_Audit	Auditor:	Rick (name of the auditor)	Time:	2018-03-14T11:00 (audit date and time)	Result   Open (audit result)		Report: 09439 (detailed problem n
	v2_Audit	Auditor:	George (name of the auditor)	Time:	2018-03-18T11:00 (audit date and time)	Result   Open (audit result)		Report: 09439 (detailed problem n
	v3_Audit	Auditor:	Padma (name of the auditor)	Time:	2018-04-15T11:00 (audit date and time)	Result   Passed (audit result)		Report: v792 (pass confirmation)

Visto così, ovviamente, non si capisce nulla. Posso addirittura mettere assieme cose dove gli attributi hanno significato diverso su ogni riga. Lo ha spiegato ma tl;dr noi useremo delle regole di mapping un po' più rifeide e chiare.

**! Va bene che è schemaless ma è bene avere un minimo di struttura comune. !**

Esistono delle API per accedere ai dati con le chiavi o con un intervallo di chiavi.

Gli elementi fondamentali erano tabelle e item, dove gli item sono caratterizzati dalla nostra chiave e un insieme di attributi.

## MAPPING FISICO: DYNAMODB [✓]

Tentiamo di trovare regole ragionevoli di corrispondenza tra i due mondi. 😊

Anche qui si parte da un'analisi dei percorsi di accesso ai dati; vogliamo fare in modo che i dati che servono alla singola applicazione siano rappresentati in un numero ridotto di record in una tabella, senza dover fare join.

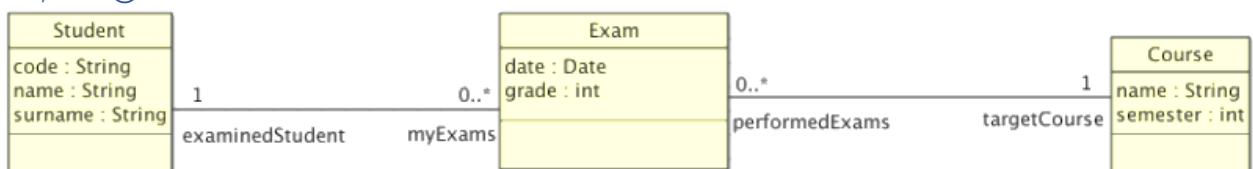
Quindi, vogliamo prepararci dei record abbastanza ricchi magari con ridondanza, che siano in grado di alimentare la singola applicazione in maniera esaustiva. Quindi, individuiamo una  $C_{AP}$  (classe principale di accesso) con una collezione che rappresenti le istanze di  $C_{AP}$ .

Similmente ad HBASE, anche qui introduciamo alcuni classifier con stereotipi specifici di questa tecnologia.

- Identificare una classe di percorso di accesso, che denominiamo  $C_{AP}$ , e rappresentarla

<pre>&lt;&lt;ItemType&gt;&gt; IT1</pre> <p>key: KT1 value: DT1</p>	<b>Itemtype:</b> Contiene chiave e valore; poi useremo una codelist per la chiave e una struttura complessa per il value.
<pre>&lt;&lt;CodeList&gt;&gt; KT1</pre> <p>&lt;pk-attribute&gt; ...</p>	<b>Tipo chiave</b> La scelta della chiave è fondamentale, perché deve guidare il 90% dell'accesso all'informazione.  Definirla come codelist ci consente di definire quale degli attributi già precisati concettualmente vogliamo usare come chiavi di accesso – eventualmente anche chiavi composte.
<pre>&lt;&lt;DataType&gt;&gt; DT1</pre> <p>attr_1 [1...*] : DT2 attr_2 [0...1] : Long attr_3 : Float ...</p>	<b>Tipo valore</b> Usiamo un classifier stereotipato in modo specifico per descrivere il valore. Possiamo avere: <ul style="list-style-type: none"> <li><b>Tipi “semplici” / scalari:</b> string, int, long, float, double, date, time, timestamp</li> <li><b>Tipi complessi</b> che possono essere <b>anche altri datatype</b>. !! Potremmo tecnicamente mettere itemtype ma nah. Alla fine, comunque, se vogliamo fare associazioni, copieremo l'identificatore,,,</li> </ul>

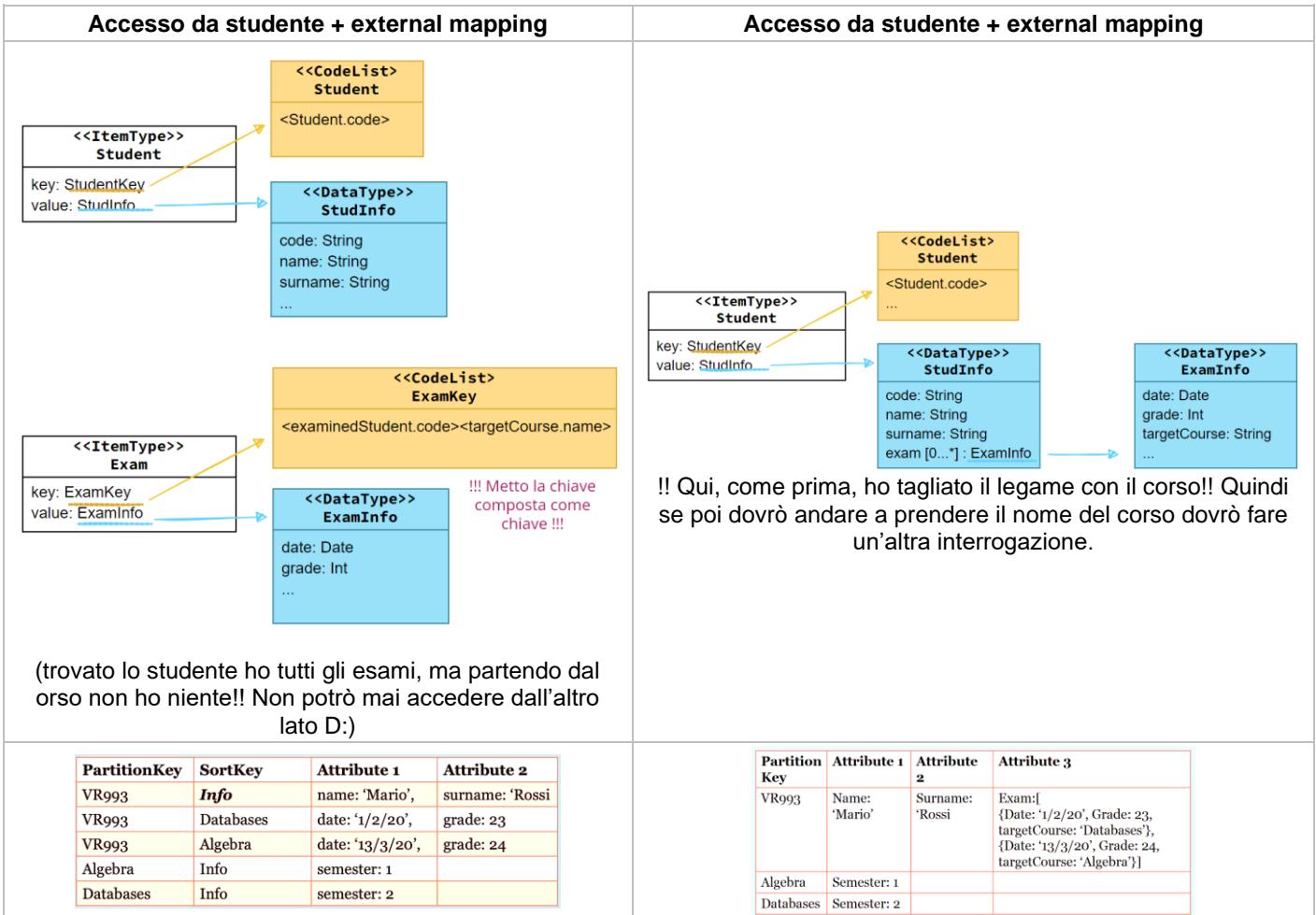
Esempio 😊



Accesso da studente	Accesso da corso
<p><b>Student</b></p> <pre> &lt;&lt;CodeList&gt;&gt; Student &lt;Student.code&gt;</pre> <p><b>Exam</b></p> <pre> &lt;&lt;CodeList&gt;&gt; Exam &lt;Exam.date&gt;</pre> <p><b>Course</b></p> <pre> &lt;&lt;CodeList&gt;&gt; Course &lt;Course.name&gt;</pre>	<p><b>Student</b></p> <pre> &lt;&lt;ItemType&gt;&gt; Student key: StudentKey value: StudInfo</pre> <p><b>Exam</b></p> <pre> &lt;&lt;ItemType&gt;&gt; Exam key: ExamKey value: ExamInfo</pre> <p><b>Course</b></p> <pre> &lt;&lt;ItemType&gt;&gt; Course key: CourseKey value: CourseInfo</pre>

- Mapping delle associazioni one to many

Anche qui, posso decidere come gestire le classi relative ad associazioni uno a molti:



### 3. Mapping delle relazioni many to many

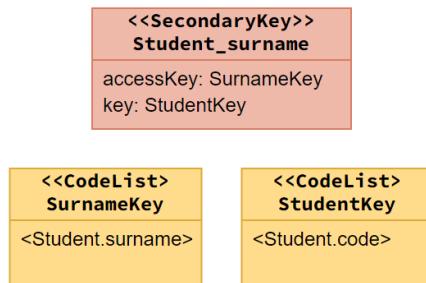
Come prima, ha senso solo il mapping interno e devo decidere su quale delle due classi farlo.

#### 4. Nota:indici secondari

DynamoDB permette di definire indici secondari, ovvero modi diversi dal primario di accedere ai dati.

Normalmente posso accedere ai dati solo via partition/sort key; se voglio accedervi diversamente dovrò costruire indici secondari, perché altrimenti l'unico approccio è fare una scansione completa.

Per aggiungere un indice secondario uso il classifier stereotipato <<SecondaryIndex>>, che definisce la chiave nuova da utilizzare e la chiave principale.

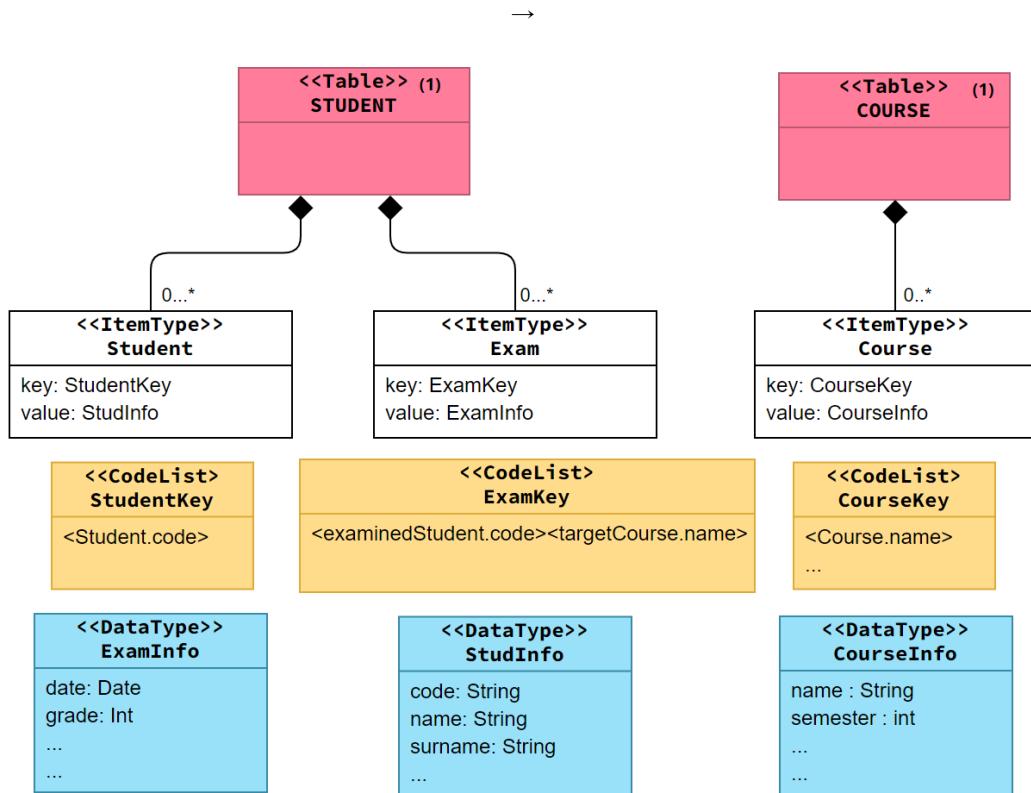


Problema degli indici secondari: **non so se è garantita la consistenza**. Potrebbe non esserlo per qualche millisecondo, e magari l'indice secondario non è aggiornato.

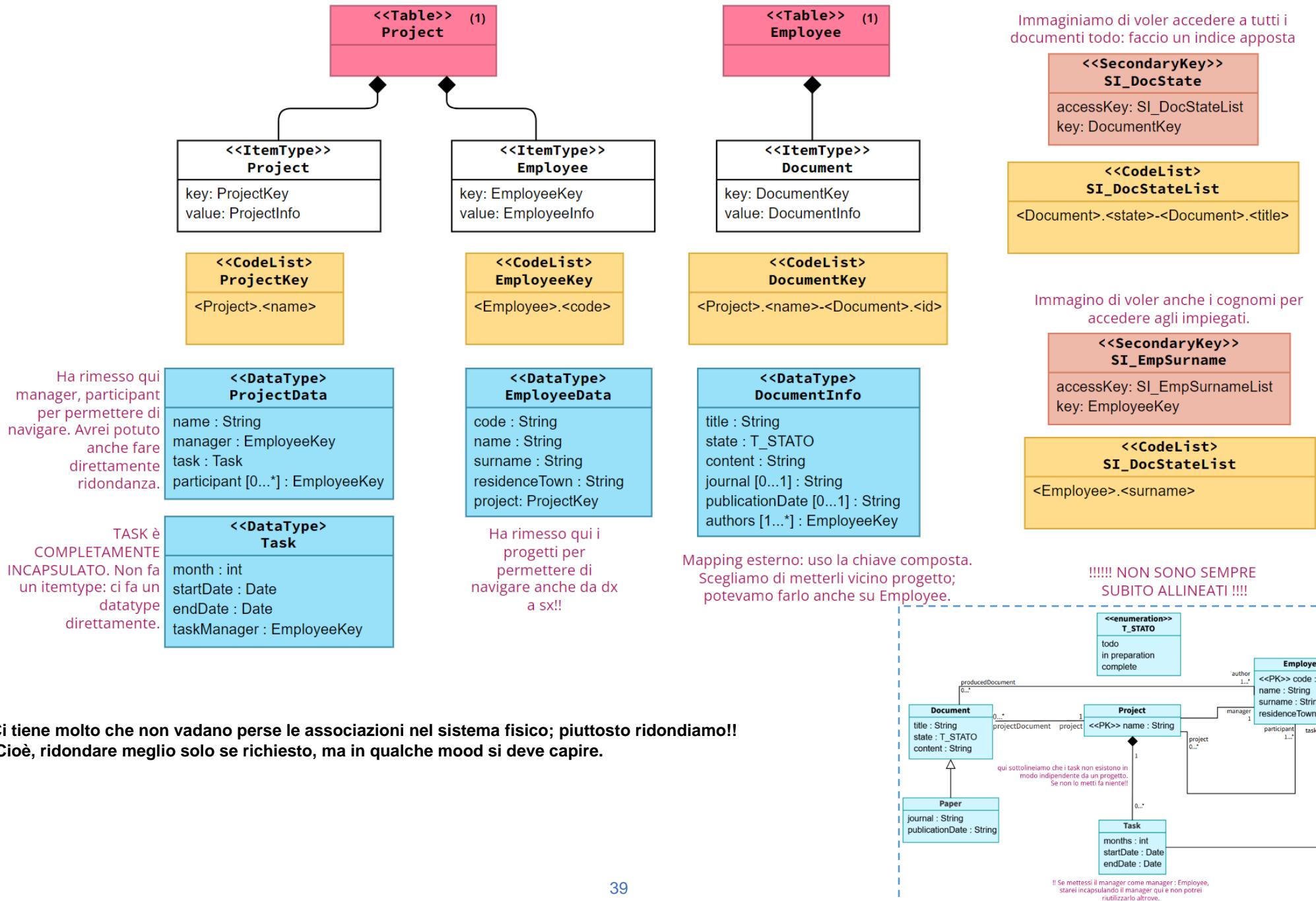
## 5. Mapping dei record definiti

- **Normalized mapping:** una tabella per ogni item type; somiglia al relazionale ed è sconsigliato.
- **Metto insieme più record**
  - **Access path based mapping:** unisco in base ai percorsi di accesso; questo significa che replico un po'
  - **Tabella unica:** tutti gli item nella stessa tabella; devo aggiungere una label con l'item type.

Di solito si procede mettendo nella stessa tabella tutti gli item che condividono partition key; non tantissime tabelle, ma qualcuna – almeno due.



## 6. Running example



## DOCUMENT DATABASES [✓]

Le tecnologie viste sono quelle con cui si è partiti, con l'obiettivo di smontare completamente il relazionale; con i document-based, invece, **si torna in direzione con un modello di dati più preciso**, pur essendo molto più flessibile del relazionale. Dunque, torniamo a trovare linguaggi di interrogazione più vicini all'SQL. Questa è la tecnologia che più spesso ha sostituito il relazionale sulla generazione di web application.

*Definizione: Document database*

Un document database è un **database nonrelazionale** che mantiene i dati come **documenti strutturati**, solitamente XML o JSON. Che sta prendendo piede. Inizialmente si usava XML, poi si passa più a JSON. Questi sistemi sono diventati il nonrelazionale per antonomasia.

C'è un nuovo modo di strutturare l'informazione. Essendo sistemi che andavano ancora sul filone della gestione di dati che sono ancora **interessati ad avere transazioni e proprietà delle transazioni**, hanno inizialmente abbandonato tutta la parte delle proprietà ACIDE per poi riprenderla.. **con accezioni e conseguenze date dal CAP theorem**.

Come per DynamoDB, anche qui si tenta di dare qualche supporto alle transazioni, seppur modesto.

La motivazione della comparsa di questi sistemi è data dal **mismatch fra ER e gli object oriented linguaggi**; l'avvento di AJAX come modello di programmazione ha dato il colpo di grazia a questa questione. Dunque, sono nati sistemi con l'obiettivo di **fornire direttamente dati direttamente in formato document**; il principe sarà MongoDB.

### XML

È proposto ancora prima del 2000 per **rappresentare un dato semistrutturato**, come evoluzione dell'HTML: i tag dell'XML non sono orientati alla definizione di come presentare l'info, ma piuttosto ad assegnarle semantica.

A inizio del 2000 sorgono una serie di protocolli di scambio dati basati su XML, come il SOAP; l'XML si ritaglia il suo ruolo prima di arrivare ai sistemi document-based come **formato per lo scambio di dati in ambito internet**, per esempio per la costruzione dei web services (=interfacce di accesso ai dati che fanno comunicare sistemi diversi; ad esempio, ESSE3 con i siti di UniVR).

Poi nascono sistemi per gettare XML nativo, ovvero che rappresentano il dato come XML e consentono di interrogarlo.

Nei sistemi relazionali Oracle ci sono millemila modi per incapsulare l'XML in attributi della tabella e funzioni per navigarli, ma non sono XML nativi. Esistono anche sistemi nativi, ma non hanno avuto tanto successo in quanto troppo verboso. L'evoluzione successiva è stata MongoDB.

Ci sono standard e tools per usarli:

- **XMLSchema** è un linguaggio formale per definire la struttura
- **XPath** e **XQuery** sono linguaggi formali per navigare e fare query su XML
- **XSLT** è un linguaggio per trasformare documenti XML in altri documenti o in altri formati.

Esistono **rappresentazioni grafiche “ad albero”** degli XML.

12/04/2022

### Punti chiave

- **Collezione di documenti**
- Dati **complessi** e a **struttura variabile**
- Ciascuna istanza di informazione è un elemento
- Un elemento può essere strutturato o testo puro
- Si può **incapsulare molto** il dato, producendo ridondanza e efficienza

### JSON

XML viene soppiantato dal JSON proprio per il fatto di essere **verboso**, soprattutto in ambito web, e fare i parser era complicato. Si propone dunque come alternativa per questo uso, e XML diventa un linguaggio per gestire informazione documentale vera e propria , come ad esempio articoli di ricerca.

Quindi, nascono nuovi formati per **gestire direttamente il dato JSON**.

#### JSON database

Un JSON database è una **collezione** (o data bucket) è un insieme di documenti che condividono un qualche scopo. Corrisponde a una tabella. Non tutti gli elementi al suo interno devono essere dello stesso tipo. Il costrutto fondamentale di questo è il documento JSON.

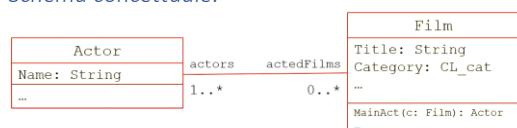
I documenti hanno struttura complessa, molto simili agli oggetti; questo è voluto, in quanto finalmente si riesce a evitare il mismatch fra un relazionale piatto e le strutture ad oggetti usate nella programmazione.

Questi sistemi sono i più carozzati per modello di dati: posso anche usarli in maniera molto simile al relazionale, avendo una collezione di documenti dove i documenti somigliano alle tuple, e soddisfare una delle forme normali del relazionale.

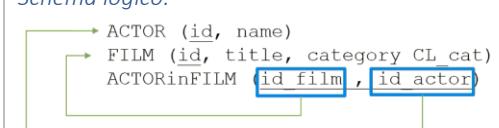
## Modellazione del dato in JSON

### Relazionale

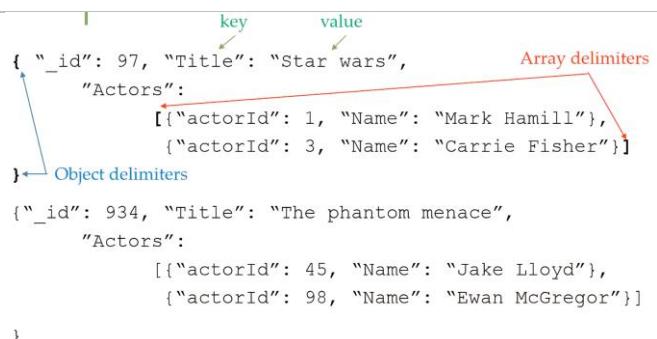
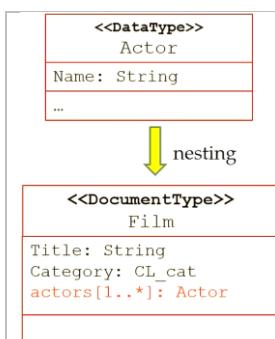
#### Schema concettuale:



#### Schema logico:



### Document based



Ci sarà una domanda su uno di questi due linguaggi che seguono!!

## DOCUMENT DATABASE COUCHDB: LINGUAGGI N1QL [✓] E MONGODB [✗]

CouchDB è un primo tentativo del 2005; memorizza documenti XML, ma poi immediatamente passa a **JSON**.

Il sistema è figlio dell'approccio di quegli anni, abbiamo anche l'inclusione dell'**approccio mapreduce** e usa l'**eventual consistency**.

Questo sistema diventa anche un progetto Apache, e va a fondersi con Membase, che abbiamo già citato per Amazon; è il sistema che fornisce MemCached, la soluzione che inizialmente sostituisce i DB pesanti con MySQL.

Con la fusione di queste due società nasce Couchbase, che introduce N1QL – un linguaggio dichiarativo ma potente. A lui pare piacere.

### N1QL

N1QL sta per “Non first normal form query language”. Un database è in forma normale se tutte le sue proprietà non possono assumere più di un valore. È un linguaggio dichiarativo che permette di eseguire query complesse. Essendo dichiarativo, ha una struttura molto simile a SQL con la SELECT...WHERE...FROM a cui siamo già abituati.

Le collezioni di partenza sono quelle dei documenti, ma chiaramente possiamo generarne altre man mano che facciamo query.

**N1QL esprime qualunque query esprimibile dal calcolo relazionale, almeno per le classi viste.**

#### 1. Banale : Trova il titolo del film con ID 97.

Posso fare una interrogazione molto banale del tutto simile a SQL: Il risultato è document based, quindi non è una tupla o una tabella ma è un documento a struttura fissa, con result e status.

```
SELECT 'Title' FROM films WHERE _id = 97; → "results": [ { "Name": "Star wars" } ],  
"status" : "success"
```

#### 2. Navigazione negli array : Trova il nome del primo attore del film nella categoria “science fiction”.

Se si tratta di un array di documenti, posso fare qualcosa di completamente analogo alla notazione degli array.

```
SELECT Actors[0].'Name' FROM films  
WHERE 'Category' = 'Science fiction'; → "results": [ { "Name": "Mark Hamill" },  
{ "Name": "Tom Holland" } ],  
"status" : "success"
```

#### 3. Quantificatore esistenziale : Trova il titolo e la categoria del film che hanno fra gli attori almeno un attore di nome “Tom Holland”.

Come in SQL, posso esprimere condizioni su interrogazioni nidificate usando operatori del tutto analoghi ai quantificatori.

```
SELECT 'Title', 'Category' FROM films  
WHERE ANY Actor IN films.Actors SATISFIES  
(Actor.'Name'='Tom Holland') END ; → "results": [ { "Title": "Spiderman",  
"Category": "Science Fiction" } ],  
"status" : "success"
```

#### 4. Quantificatore universale : Trova il titolo dei film che durano più di 130 minuti, insieme al nome degli attori.

Con UNNEST posso riportare cose innestate al primo livello. Attenzione: questa condizione varierà in base alla condizione prima dell'nnest: nella clausola FROM sto generando coppie del tipo film1-attore1, film1-attore2, film1-attore3... . Somiglia molto al relazionale, ma attenzione che non è il prodotto cartesiano! L'unnest riporta al primo livello, e questo serve sia a esprimere condizioni che a produrre una struttura diversa.

```
SELECT x.'Title', y.'Name' FROM films x  
UNNEST x.Actors y → "results": [ { "Title": "Spiderman",  
"Name": "Tom Holland" },  
{ "Title": "Spiderman",  
"Name": "Zendaya" } ],  
"status" : "success"
```

#### 5. JOIN. : Trova il titolo dei film che sono stati selezionati per il premio.

Siamo comunque in un non-relazionale: ne deriva che, pur potendolo esprimere, le prestazioni saranno pessime.

Se uso ON KEYS usa automaticamente l'id.

```
SELECT x.'Title' FROM award  
JOIN films x ON KEYS award.filmID; → "results": [ { "Title": "Star wars" } ],  
"status" : "success"
```

## MongoDB [X]

---

Non sistemo questi appunti perché faccio N1QL. As I said sorry guys feel free to add stuff.

Negli anni 200 nasce Mongo DB, immediatamente per JSON. Rappresenta in maniera binaria, ovvero in **BSON**, orientato per i big data. Al contrario di CouchBase, il linguaggio è **procedurale** e assomiglia a un'algebra.

MongoDB ha molto successo e diventa il sistema di riferimento per l'architettura delle applicazioni web, sostituendo mySQL.

È lo stesso modello di CouchBase: collezioni di documenti da gestire.

Il linguaggio è in JS, ma si riesce a interagire anche in maniera più facile e diretta (un po' come PGADMIN per postgres). La sintassi è basata su due operatori:

find()

Consente di lavorare su una singola collezione e produrre un risultato.  
Consente di fare selezioni e proiezioni.

aggregate()

Consente di definire una sequenza di operazioni da applicare su una collezione, eventualmente coinvolgendo più di una.

L'idea è di partire da una collezione di documenti, e mischiando questi due operatori si ottiene una nuova versione.

Query semplici usando la find

Trova tutti i film	<code>db.films.find({})</code>
Trova i film della categoria "Science fiction"	<code>db.films.find( {Category:"Science Fiction"} )</code>
Trova i film della categoria "Science fiction" e di lunghezza 125 minuti	<code>db.films.find( {category:"Science Fiction",Length:125} )</code>
Trova i film della categoria "Science Fiction" oppure "Horror"	<code>db.films.find( {Category:{\$in:["Science fiction","Horror"]}} )</code>

I join sono stati inseriti solo in un secondo momento. Si fa cambiando la funzione: al posto di find, uso aggregate.

[...]

## ANALISI DEI LINGUAGGI DI INTERROGAZIONE DEI SISTEMI DOCUMENT BASED [X]

Non ho avuto tempo di sistemare, again, scusate. Feel free to add urself

### Schema

- MEDICAL\_CENTER(CenterCode, Name, Municipality, Region, Lat, Long);
- TEST(Date, Patient, Type, Time, Processed, Result, Center)
- PATIENT(PatCode, Age, Nationality, DateOfBirth, ResMunicipality, ResRegion)

I JSON sono divisi in due buckets:

Centers	Patients
<pre>{   "_id": 1, "Code": "C1", "Name": "Centro VR1",   "Municipality": "Verona", "Region": "Veneto",   "Lat": 45, "Long": 13.1 }  {   "_id": 2, "Code": "C2", "Name": "Centro VR2",   "Municipality": "Verona", "Region": "Veneto",   "Lat": 45, "Long": 13.2 }  {   "_id": 3, "Code": "C3", "Name": "Centro VII",   "Municipality": "Vicenza", "Region": "Veneto",   "Lat": 44.9, "Long": 14 }</pre>	<pre>{   "_id": 13, "Code": 12392, "Age": 34, "Nationality": "ita",   "DateOfBirth": "10-10-1087",   "ResMunicipality": "VR", "ResReg": "Veneto",   "Tests": [{"_id": 103, "Date": "1-1-2021", "Time": "07:03",   "Processed": true, "Result": "positive", "Center_id": 1,   "Type": "molecular swab"}]  {   "_id": 123, "Code": 12392, "Age": 61, "Nationality": "ita",   "DateOfBirth": "12-12-1060",   "ResMunicipality": "VI", "ResReg": "Veneto",   "Tests": [{"_id": 133, "Date": "1-3-2021", "Time": "11:13",   "Processed": true, "Result": "negative", "Center_id": 3,   "Type": "rapid test"}, {"_id": 412, "Date": "1-4-2021", "Time": "11:43",   "Processed": true, "Result": "negative", "Center_id": 2,   "Type": "molecular swab"}] }</pre>

Basically abbiamo schiacciato i test dentro i pazienti.

### CouchDB e N1QL

Q1: centri medici del veento situati né a verona né a venezia riportando il codice, il nome e le coordinate del centro.

RC:  $\{c, n, lat, long \mid \exists m, r. (MEDICAL\_CENTER(c, n, m, r, lat, long) \wedge r \neq 'Veneto' \wedge \neg(m = 'Verona' \vee m = 'Venice'))\}$

```
SELECT Code, Name, Lat, Long
FROM Centers
WHERE Region = "Veneto" AND
NOT (Municipality = "Verona" OR Municipality = "Venice");
```

```
SELECT Code, Name, Lat, Long
FROM Centers
WHERE Region = "Veneto" AND
Municipality NOT IN ["Verona", "Venice"];
```

Q2: trovare i pazienti con età maggiore di 55, nazionalità inglese o irlandese, riportando il codice la municipalità di residenza e la nazionalità del paziente

RC:  $\{c, rm, n \mid \exists a, db, r. (PATIENT(c, a, n, db, rm, r) \wedge a > 55 \wedge (n = 'english' \vee n = 'irish'))\}$

N1QL:

```
SELECT Code, ResMunicipality, Nationality
FROM Patients
WHERE Age > 55 AND
Nationality IN ["english", "irish"]
```

Q3: trova tutti i test del paziente che viene dalla sicilia riportando nel risultato patient code, risultato e data del test.

RC:  $\{(c, r, d \mid \exists a, n, db, rm. (PATIENT(c, a, n, db, rm, 'Sicily')) \wedge \exists ty, tm, p, ct. (TEST(d, c, ty, tm, p, r, ct)))\}$

N1QL:

```
SELECT pat.Code, t.Result, t.Date
FROM Patients AS pat
UNNEST pat.Tests AS t
WHERE pat.Region = "Sicily"
```

Usiamo l'UNNEST proprio per smontare l'incapsulamento. Qui poteva fare a meno ma poi servirà, so.

*Q4: trova i pazienti da lombardia che nella stessa giornata hanno fatto un test rapido e uno molecolare, riportando nel risultato codice e nazionalità del paziente assieme alla data dei test.*

Condizione da esprimere sulla tabella test

```
RC: {c, n, d |  $\exists a, db, rm.(\text{PATIENT}(c, a, n, db, rm, 'Lombardy') \wedge$ 
 $\exists tm, p, r, ct.(\text{TEST}(d, c, 'rapid test', tm, p, r, ct) \wedge$ 
 $\exists tm', p', r', ct'.(\text{TEST}(d, c, 'molecular swab', tm', p', r', ct')))}$ }
```

N1QL:

```
SELECT pat.Code, pat.Nationality, t1.Date
FROM Patients AS pat
UNNEST pat.Tests AS t1 UNNEST pat.Tests AS t2
WHERE pat.Region = "Lombardy" AND t1.Date = t2.Date AND
t1.Type = 'rapid test' AND t2.Type = 'molecular swab'
```

Poiché non sto facendo il relazionale, e non ho join, likely ci mette unbotto di meno 😊😊😊

*Q5: find for each patient, if they exist, the pairs of consecutive positive tests of type'molecular swab' reporting the code of the patient and the date of both tests (use only the attribute "Date" for ordering the tests on the temporal axis)*

```
RC: {c, d1, d2 |  $\exists a, n, db, m, r.(\text{PATIENT}(c, a, n, db, m, r)) \wedge$ 
 $\exists tm, p, ct.(\text{TEST}(d_1, c, 'molecular swab', tm, p, 'true', ct) \wedge$ 
 $\exists tm_2, p_2, ct_2.(\text{TEST}(d_2, c, 'molecular swab', tm_2, p_2, 'true', ct_2) \wedge$ 
 $d_1 < d_2 \wedge$ 
 $\neg \exists tm_3, p_3, ct_3.(\text{TEST}(d_3, c, 'molecular swab', tm_3, p_3, 'true', ct_3) \wedge$ 
 $d_1 < d_3 \wedge d_3 < d_2))}}$ }
```

N1QL:

```
SELECT pat.Code, t1.Date, t2.Date
FROM Patients AS pat
UNNEST pat.Tests AS t1 UNNEST pat.Tests AS t2
WHERE t1.Date < t2.Date AND t1.Type = 'molecular swab' AND
t2.Type = 'molecular swab' AND t1.Result = 'positive' AND
t2.Result = 'positive'
NOT ANY tx IN pat.Tests
SATISFIES t1.Date < tx.Date AND tx.Date < t2.Date END
```

*Q6: find the centers, that has done no one test on '1-1-2021' reporting in the result he code of the center and the tests done on '21-5-2021'.*

La collezione dei test è dentro il paziente in N1QL, quindi ogni volta che voglio i test mi tocca partire da paziente.

```
{n, c, p1, ty1, tm1, r1 |  $\exists m, r, lt, lg.(\text{MEDICAL_CENTER}(c, n, m, r, lt, lg) \wedge$ 
 $\neg \exists p, ty, tm, pr, r.(\text{TEST}('2021 - Jan - 1', p, ty, tm, pr, r, c)) \wedge$ 
 $\exists pr_1.(\text{TEST}('2021 - May - 21', p_1, ty_1, tm_1, pr_1, r_1, c)))}$ }
```

N1QL:

```
SELECT cen.Code, te AS tests
FROM Patients AS pat UNNEST pat.Tests te
JOIN Center AS cen ON KEYS te.Center_id
WHERE NOT cen._id
WITHIN (SELECT t.Center_id
FROM Patients UNNEST Patients.Tests t
WHERE t.Date = '1-1-2021')
AND te.Date = '21-5-2021'
```

! Il join è fatto su una chiave quindi in realtà aggancio sempre e solo 1 cosa.

Con within Siamo in grado di simulare esistenziale e esistenziale negato, quindi la potenza espressiva è massima! È un linguaggio di interrogazione come si deve.

!! POSSIBILE DOMANDA DELLO SCRITTO: rappresentare una delle interrogazioni in N1QL o MongoDB. N1QL è più facile.

## MongoDB

Q1: centri medici del veento situati né a verona né a venezia riportando il codice,il nome e le coordinate del centro.

```
db.Centers.find(
  { Region: "Veneto",
    {$not: { $or: [{Municipality: "VR"},{Municipality: "VE"}]}}
  },
  {Code: 1, Name: 1, Lat: 1, Long: 1})
```

Q2

```
db.Patients.find(
  { Age: {$gt: 55},
    Nationality: { $in: ["english", "irish"] }
  },
  {Code: 1, ResMunicipality: 1, Nationality: 1})
```

Q3

```
db.Patients.find( {Region: "Sicily"},
  {Code: 1, "Tests.Result": 1, "Tests.Date": 1})
```

Q5

Non riesco più a usare find perché devo negare un esistenziale → non riesco a scrivere l'anegazione dell'esistenziale. Quindi bisogna cercare un approccio diverso,,,

Comincia la diatriba fra far fare il lavoro al linguaggio di interrogazione o da codice. Generando tutti i risultati intermedi che voglio posso sempre avere il risultato, ma è troppo complicato e dipende dalla mia abilità di usare le cose :)

Questo è *particolarmente* complicato in questo esempio. Tocca usare aggregate.

Q6

MongoDB: using the aggregate function with a pipeline we can obtain this:

```
db.Patients.aggregate([
  { $unwind: "$Tests" },
  { $out : "Tests" }

  db.Patients.aggregate([
    { $unwind: "$Tests" },
    { $match : { "Tests.Date": "21-05-2021" } },
    { $out : "TestsOn210521" }
  ]

  db.Centers.aggregate([
    { $lookup : { from: "Tests", localField: "_id",
      foreignField: "Tests.Center_id", as: "TestsAtCenter" },
      { $lookup : { from: "TestsOn210521", localField: "_id",
        foreignField: "Tests.Center_id", as: "TestsOn210521" },
        { $match : { $nor: [ {"TestsAtCenter.Date": "1-1-2021" } ] } }
    }
  ]);
])
```

Similar to UNNEST

↑ Similar to not exists one test on 1-1-2021

→ sto togliendo i test da paziente e rimettendoli nel centro.

Il meccanismo per fare il not è che metto un \$match on il not su una proprietà espressa,,,

## MAPPING FISICO: DOCUMENTDB [✓]

L'idea generale è che dobbiamo decidere quali delle classi concettuali rappresentare come bucket di documenti.

Sicuramente definiamo una collezione di documenti per la nostra classe di accesso. Poi possiamo definire la struttura del document type; possiamo descriverla irettamente, dato che non abbiamo solo il key-value, ed eventualmente encapsulare altre classi se necessario.

<pre>&lt;&lt;DocumentType&gt;&gt; DocT1</pre> <pre>attr_1 [1...*] : DT2 attr_2 [0...1] : Long attr_3 : Float</pre>	<p>Usiamo lo stereotipo DocumentType, all'interno del quale metteremo direttamente gli attributi di primo livello.</p>
<pre>&lt;&lt;DataType&gt;&gt; DT2</pre> <pre>attrDT_1 : Int attrDT2_2 [0...1] : Long ...</pre>	<p>Per tutti gli attributi che encapsulano struttura complessa, definiamo un datatype.</p>

### 1. Traduco le classi principali

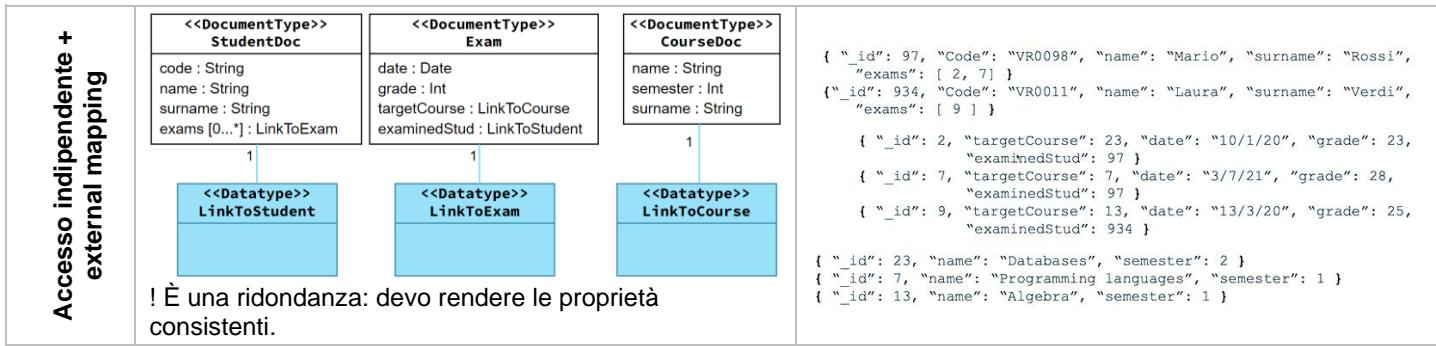
Accesso da studente	Accesso da corso
<pre>&lt;&lt;DocumentType&gt;&gt; StudentDoc</pre> <pre>code : String name : String surname : String</pre>	<pre>&lt;&lt;DocumentType&gt;&gt; CourseDoc</pre> <pre>name : String semester : Int</pre>

### 2. Associazioni uno a molti

Per tutte le altre classi ho diverse possibilità:

- **Internal mapping:** encapsulo e metto tutto dentro nel document type di C<sub>AP</sub>, e definisco un nuovo datatype che definisce le istanze encapsulate
- **External mapping con accesso da CAP:** dedico un documento alle istanze di C, e implemento la struttura di C senza il legame verso C<sub>AP</sub>; posso navigare verso C solo da CAP
- **External mapping con accesso indipendente:** posso rappresentare nei documenti che istanziano C anche il legame inverso verso C<sub>AP</sub>, ovvero posso immaginare di navigare anche partendo da C verso C<sub>AP</sub>; C diventa una specie di C<sub>AP</sub>.

<p><b>Accesso da studente + internal mapping</b></p> <pre> &lt;&lt;DocumentType&gt;&gt; StudentDoc code : String name : String surname : String exams [0..*] : Exam  &lt;&lt;DocumentType&gt;&gt; Exam date : Date grade : Int targetCourse : LinkToCourse  &lt;&lt;DocumentType&gt;&gt; CourseDoc name : String semester : Int exams [0..*] : Exam   </pre> <p>Specifichiamo che è un link a un corso (e non un generico intero) usiamo questa notazione.</p>	<pre> {   "_id": 97, "Code": "VR0098", "name": "Mario", "surname": "Rossi",   "exams": [     {"targetCourse": 23, "date": "10/1/20", "grade": 23},     {"targetCourse": 7, "date": "3/7/21", "grade": 28}   ] }  {   "_id": 934, "Code": "VR0011", "name": "Laura", "surname": "Verdi",   "exams": [     {"targetCourse": 13, "date": "13/3/20", "grade": 25}   ] }   </pre>
<p><b>Accesso SOLO da studente + external mapping</b></p> <pre> &lt;&lt;DocumentType&gt;&gt; StudentDoc code : String name : String surname : String exams [0..*] : LinkToExam  &lt;&lt;DocumentType&gt;&gt; Exam date : Date grade : Int targetCourse : LinkToCourse  &lt;&lt;DocumentType&gt;&gt; CourseDoc name : String semester : Int surname : String  &lt;&lt;Datatype&gt;&gt; LinkToExam targetCourse : LinkToCourse  &lt;&lt;Datatype&gt;&gt; LinkToCourse targetCourse : LinkToCourse   </pre>	<pre> {   "_id": 97, "Code": "VR0098", "name": "Mario", "surname": "Rossi",   "exams": [ 2, 8 ] }  {   "_id": 934, "Code": "VR0011", "name": "Laura", "surname": "Verdi",   "exams": [ 9 ] }  {   "_id": 2, "targetCourse": 23, "date": "10/1/20", "grade": 23 } {   "_id": 8, "targetCourse": 7, "date": "3/7/21", "grade": 28 } {   "_id": 9, "targetCourse": 13, "date": "13/3/20", "grade": 25 }  {   "_id": 23, "name": "Databases", "semester": 2 } {   "_id": 7, "name": "Programming languages", "semester": 1 } {   "_id": 13, "name": "Algebra", "semester": 1 }   </pre>



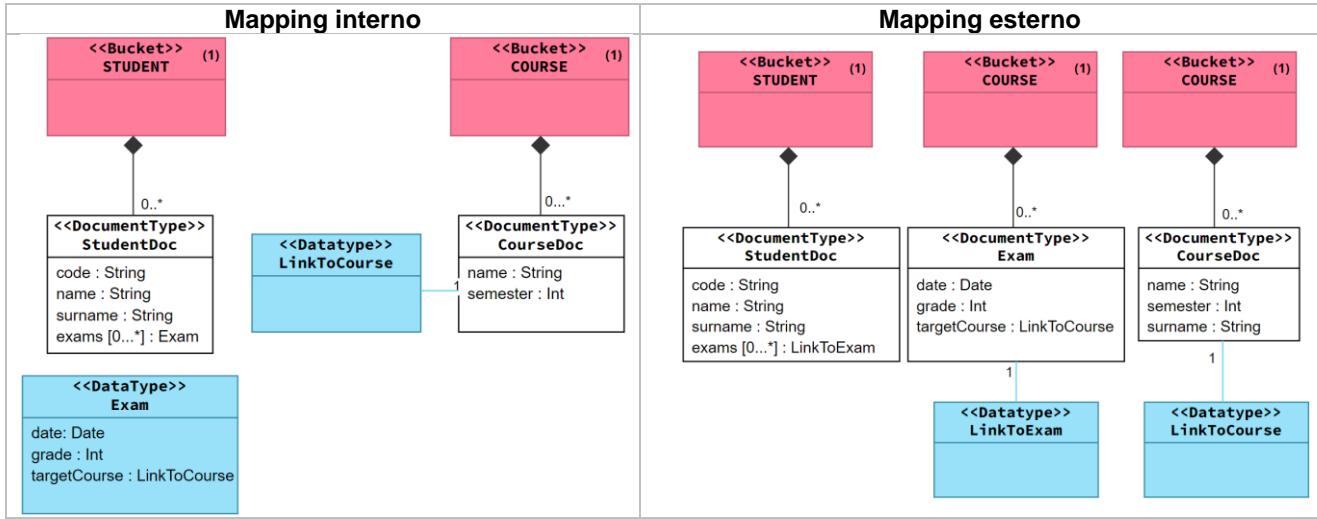
### 3. Associazioni molti a molti

Per le molti a molti:

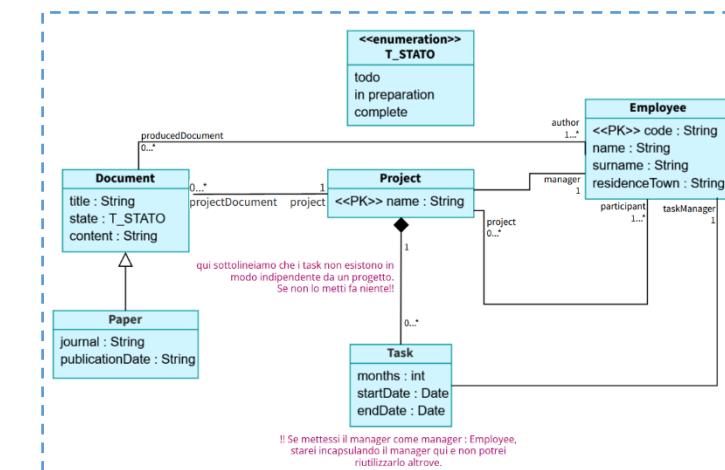
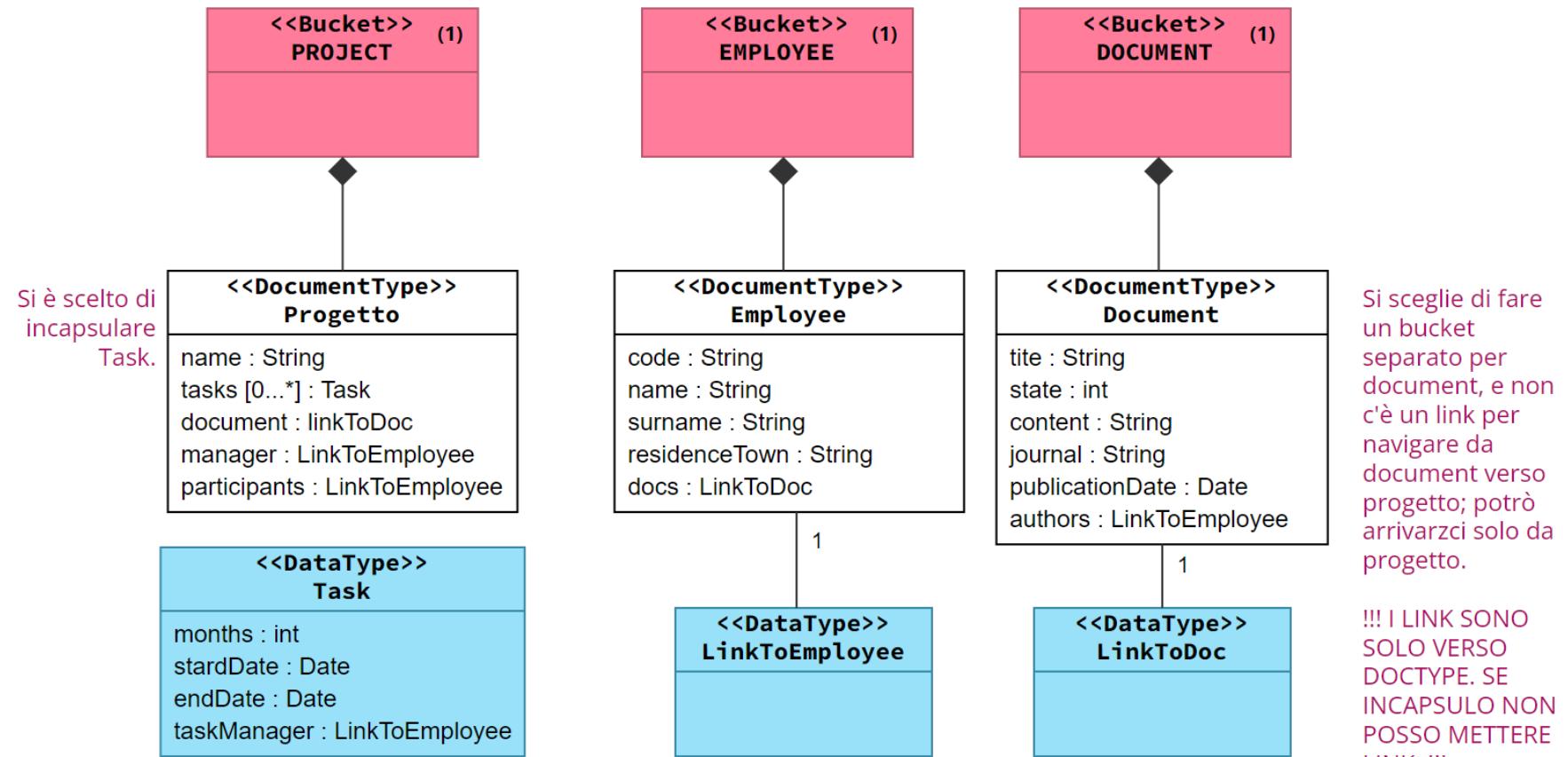
- Mapping interno in CAP oppure CAP' oppure entrambe, rappresentando solo l'id
- Mapping esterno: come nel relazionale.

### 4. Mapping nei bucket

Ho una sola scelta: un bucket per ciascun Document Type definito nel modello.



## Running example



## COLUMNAR DATABASES [✓]

Sono cose che nascono intorno agli anni 2000 per servire quelle applicazioni che fanno analisi per il supporto alle decisioni, ovvero data warehousing o Data Mining™.

### OLTP

Tipicamente, nel relazionale, **una riga corrisponde a un'istanza**; di solito c'è una corrispondenza precisa fra le righe di una tabella e un oggetto. Siamo abituati a gestire la base di dati come se fosse una collezione di righe da aggiugere/togliere/modificare, e interroghiamo per righe. Questo è l'**OLTP (On Line Transaction Processing)**.

### Data Warehousing e OLAP

Successivamente, c'è un ulteriore passaggio che consiste nel realizzare che, finito il processo (per esempio di vendita), i dati potrebbero essere usati per fare analisi o supporto alle decisioni; nasce l'idea di **data warehousing** o **data analytics**.

Questo processo **lavora per colonne**: tende a considerare tutte le istanze dell'informazione e analizzare una caratteristica – che è una colonna (ad esempio, "data della fattura"). Nascono dunque sistemi orientati all'analisi per colonna, ma in tempi più recenti anche essi diventano inutili a causa della mole di dati.

Per rendere efficiente questo tipo di applicazione rispetto all'attività normale del sistema, si pensa inizialmente di separarli: **faccio una copia dell'info storicizzata in un altro DB** e uso quello per fare le decisioni. Di qui, dunque, anche una ristrutturazione al dato (durante la copia) per semplificare le operazioni. Nasce dunque lo **Star Schema**, schema tipico d'un data warehouse.

#### *Data Warehousing*

Il data warehouse è un **processo** che ha il fine di estrarre ed integrare dati storici da sistemi diversi e non omogenei, per supportare il **sistema decisionale** della compagnia.

È un approccio nel quale voglio andare in parallelo al OLTP per elaborare dell'informazione storicizzata.

Chiamiamo questa attività **OLAP (On Line Analytic Processing)**.



Difficoltà:

- Enorme mole di data
- DB diversi ed eterogenei
- Dati da classificare e raggruppare, oltre che da integrare fra loro.

Le sue caratteristiche sono:

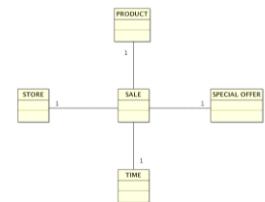
- **Orientato al soggetto**: mi interessa partire dai concetti (? Credo ? Belu non ti si capisce)
- **Integrato**: c'è bisogno di fare integrazione, quindì mi servirà uno schema condiviso per ripulire e riunire schemi diversi (es. indirizzo rappresentato in maniera diversa).
- **Correlato al tempo**: è una delle dimensioni per analizzare il dato
- **Persistente**: non genero il DW al volo! Non ne faccio una copia! Lo rendo persistente per poter farci operazioni sopra. Inoltre, non faccio aggiornamenti; tipicamente si rifà da capo.
- **Di supporto alle decisioni**

L'operazione di estrazione fa anche una pulitura/estrazione, per poi generare il warehouse

### Schemi

Sono tipicamente a stella/fiocco di neve, ovvero una tabella centrale che si correla con i fatti. Ogni data warehouse si divide in più parti che sono dette **DATA MART**, che indirizzano uno specifico obiettivo di analisi.

Al centro della stella c'è una collezione di dati, che rappresentano i **FATTI**. Sui margini della stella, invece, troviamo le **DIMENSIONI** di analisi; ciascuna dimensione potrebbe poi avere una struttura a sua volta gerarchica (es. nel caso del tempo, giorno-ora-minuto...), generando uno schema a **fiocco di neve**.



## Interfacce grafiche

In questi anni nascono anche le prime interfacce grafiche che permettono di analizzare dei dati, appoggiandosi su più sistemi con questo approccio. L'obiettivo è generare report e fare interrogazioni o diagrammi aggregando i dati in modo diverso. La funzione di aggregazione è uno dei principali operatori.

Le interfacce possono essere riassunte in modo molto spartano come di sotto; permettono di lavorare su uno schema a stella avendo una colonna per ogni dimensione di analisi.

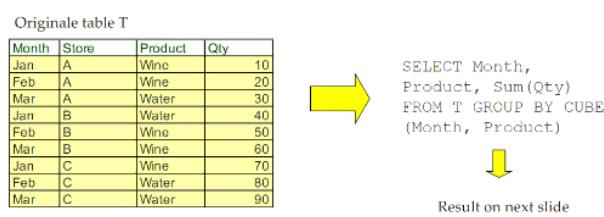
dim1	dim2	fact (F.c)	fact (F.d)
current values	current values		
selection	selection		
group by	group by	OP1 <sub>Aggr</sub>	OP2 <sub>Aggr</sub>

Alla fine, per potere espressivo, è del tutto equivalente a SQL.

## MDA tools

Rappresentiamo i dati come cubo di Rubik a n dimensioni dove potrei tagliar fette, tagliare una dimensione, ...le operazioni sono tutte visualizzabili sul cubo.

Viene esteso anche SQL, che infatti ha delle operazioni basate sulla variante CUBE.



Con un group by normale avrei tutte le combinazioni; nel cube viene anche generato il gruppo che da una parte ha una delle dimensioni e dall'altra tuttte. (? Mica ho capito belu)

Ci sono altre operazioni:

- **Slice and dice:** selezione di cubo secondo certe condizioni
- **Drill down:** aggiungo una dimensione
- **Roll-up:** tolgo una dimensione

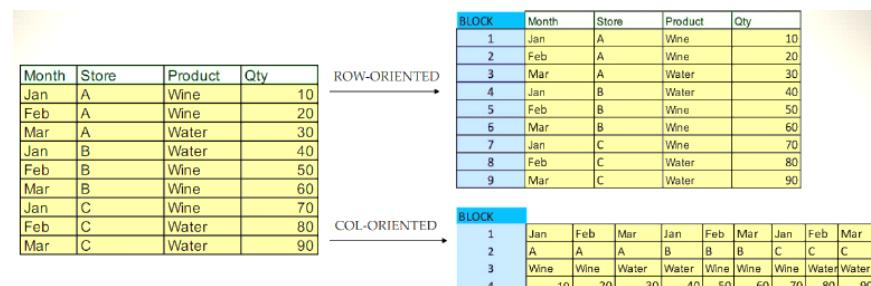
Fare queste operazioni su certe dimensioni (grandi) manda il sistema in crisi, nonostante eventuali ottimizzazioni... Dunque ci si orienta su un altro sistema, al solito 😊

## Columnar database

Il columnar approach decide di lavorare sulle colonne. L'idea è che la tabella a livello logico resta lo strumento di lavoro, ma dietro le quinte le tabelle sono smezzate in modo diverso.

### Vantaggi:

- Se devo fare **operazioni su colonna** le ho tutte su un blocco, e dunque sono molto più veloci
- Il fatto di organizzare per colonna mi aiuta a **comprimere meglio il dato** (sono più omogenei e comprimibili). La penalità è che se devo lavorare per riga (e prima o poi, se uso anche OLTP, succede) è un macello.

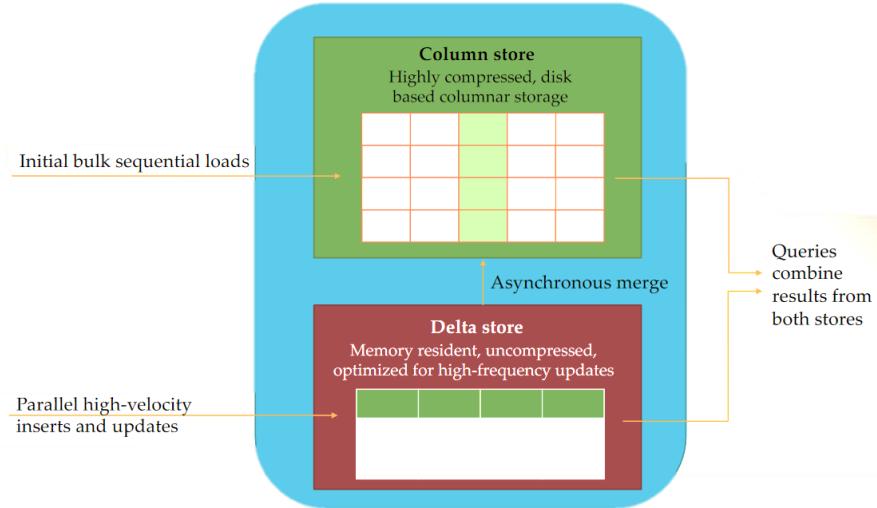


### Svantaggi:

- Se uso questo DB anche per l'OLTP, significa che prima o poi mi servirà accedere alla riga. Di conseguenza:
  - Se lo istanzio oggi volta è tutto ok, perché tanto non voglio aggiornare!

- Se invece voglio usarlo in sé, allora devo trovare degli escamotage 😞 Essa è il **delta store**, ovvero le modifiche non sono salvate subito; sono memorizzate a parte in un delta store che sarà integrato la notte/ più tardi. Tipicamente il delta store è piccolo, quindi può addirittura stare in memoria principale.

Prima del merge tutte le query devono considerare sia il delta che la parte consolidata nel db.



## Regioni e proiezioni

Spesso i db colonnari organizzano una colonna in un blocco, ma in generale posso anche organizzare le informazioni accoppiando le colonne in diverse porzioni di memoria in modo che le colonne usate insieme siano vicine. Per fare ciò si introduce il concetto di regione. Tipicamente potremmo avere una colonna in ogni regione, o tutte le colonne in una regione, ma anche raggruppamenti diversi.

In generale, invece che definire indici definiamo regioni e raggruppamenti di colonne.

In generale c'è sempre la superproiezione – ovvero la proiezione che contiene tutte le colonne, mentre possiamo aggiungere a seconda delle mie esigenze altre proiezioni aggiuntive che permettono di accedere velocemente a ciascun insieme di colonne.

### Esempi

Original Data				→	Super-projection
Region	Cust	Prod	Sales		Region A B D C A Cust G C F C R Prod C C D A B Sales 77 73 43 23 5
A	G	C	77		
B	C	C	73		
D	F	D	43		
C	C	A	23		
A	R	B	5		

Projection 1					
Cust	C	C	F	G	R
Sales	73	23	43	77	5

Projection 2					
Cust	C	C	F	G	R
Sales	73	23	43	77	5

## Esempi di sistemi

- Sybase IQ**: è il primo. È degli anni 90
- C-store**: paper di stonebraker sul cambio di visione. È un prototipo, poi diventa vertica
- Vertica**, anche parquet, vector wise, monetdb

## DATABASE A GRAFO [✓]

[NON CHIEDE LE INTERROGAZIONI] L. 15 – 03/05/2022

Motivazioni dietro questi sistemi

In tutte le tecnologie già viste, il modello di dati (implicito o esplicito) è una collezione di istanze di informazione. L'enfasi è più sulle istanze e meno sulle relazioni fra istanze.

Al contrario, i sistemi **graph-based** o **graph-oriented** è quello di dare **maggior enfasi ai legami**. I legami diventano il principale elemento di informazione da trattare; questo significa che si può usare un grafo per rappresentare l'informazione.

I benefici sono che:

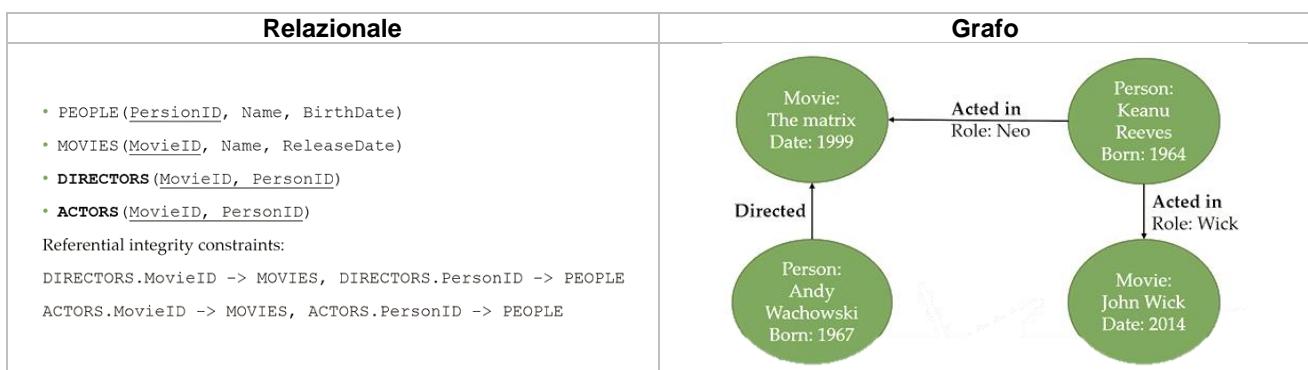
- I legami sono sempre **esplicativi**
- I legami non sono **mai tagliati**: posso sempre navigare fra i dati – non solo “ad un passo”, ma anche con percorsi molto lunghi.

Definizione: Grafo

La teoria dei grafi definisce un grafo come una struttura algebrica composta da:

- Un insieme di **vertici**  $V$
- Un insieme di **archi**  $E \subseteq V \times V$
- In alcuni casi nodi e vertici hanno delle **etichette**.

Esempio:



Querying: RDBMS

In relazionale, posso fare query su un grafo facendo una serie di join che mi permetta di estrarre le info agganciando istanze di info fra loro. Il problema è che in SQL non ho un modo per reincludere ricorsivamente un certo numero di join finché mi ricollego; è necessario, quindi, specificare la profondità di navigazione a priori.

```
SELECT p2.Name, m1.Name
FROM PEOPLE p1 JOIN ACTORS a1 ON (p1.PersonID = a1.PersonID)
JOIN MOVIES m1 ON (a1.MovieID = m1.MovieID)
JOIN ACTORS a2 ON (a2.MovieID = m1.MovieID)
JOIN PEOPLE p2 ON (p2.PersonID = a2.PersonID)
WHERE p1.Name = 'Keanu Reeves'
```

Resource Description Framework (RDF)

Sempre negli anni 2000 c'è l'esplosione del cosiddetto web semantico. L'idea è che voglio uno strumento per gestire una grande mole di dati molto destrutturati, sui quali voglio aggiungere significato/struttura facendo dell'analisi intelligente. Non è nel filone di basi di dati, perché tipicamente questo è sempre stato in ambito di AI. La descrizione dell'informazione avviene a tutti i livelli: ho un unico livello con tutto mescolato.

È un approccio formale diverso che parte dalle logiche descrittive e arriva alle ontologie. L'informazione che descrivo con questo approccio è sempre fatta da terne entity : attribute: value.

Nell'esempio precedente avrei qualcosa del tipo

```
TheMatrix: is :Movie →
Keanu: is :Person
Keanu: starred in :TheMatrix
```

Processing del grafo

Se in relazionale ho problemi man mano che trasverso il grafo, qui ho una navigazione molto facilitata, avendo legami sempre diretti.

Resta qualche problema:

- Nasce come **non distribuito**, quindi il passaggio al distribuito è complicato e il parallelismo non è banale
- **Tradurre un relazionale** in grafo è complicato. (però ci sono repurpose di altri sistemi già fatti, tipo in hadoop)

Inoltre, abbiamo problemi a rappresentare DB tradizionale. Dunque non sono in competizione con RDBMS; sono usi diversi.

## Neo4J

---

È uno dei sistemi più usati per questo tipo di database. Usa l'approccio di nodi e dati con attributi attaccati

- Può supportare grafi molto grandi, e supporta anche le transazioni acide.
- Viene un po' prima dai nuovi sistemi, quindi non è troppo orientato al distribuito.
- Ha un linguaggio di query dichiarativo; qualche esempio

Find the nodes that have a property name equal to "Keanu Reeves":

```
MATCH (k: Person {name: "Keanu Reeves"}) RETURN k;
```

Find the name of the director of the movie "Matrix":

```
MATCH (m: Movie {title: "The Matrix"})<-[:DIRECTED]-(d) RETURN d.name
```

Find all actors who have ever acted in the same movie as Keanu Reeves.

```
MATCH (k: Person {name: "Keanu Reeves"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(y)
RETURN y.name
```

## INTERNAL: DISTRIBUTED PATTERN [✓]

Con **internals** intendiamo tutte le tecniche e metodi implementate dai nuovi sistemi – consideriamo MongoDB per document-based, HBase e DynamoDB/Cassandra – per capire come cambia internamente il database “dietro le quinte”.

Nel relazionale, il primo problema riguarda la traduzione di un'espressione in linguaggio dichiarativo in un piano di esecuzione, con delle tecniche di ottimizzazione. Questa parte è poco sviluppata nei nuovi sistemi:

- Alcuni sistemi **non hanno proprio linguaggio dichiarativo**, perché il dato è talmente complicato o grande che solo il programmatore sa come deve essere 😊
- Altri sistemi hanno questo livello di ottimizzazione sviluppato dopo.

Inoltre, un'altra questione è quella dell'esecuzione concorrente garantendo le proprietà delle transazioni. Tutto questo salta nei nuovi sistemi!

### Motivazioni: perché architetture distribuite?

La motivazione è sempre la solita: necessità delle applicazioni (web) di gestire moli di dati sempre più grandi, che richiede di essere scalabili. Le soluzioni sono due:

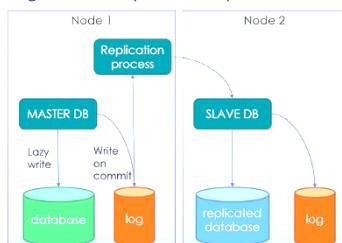
- **Scaling up**: non funziona
- **Scaling out**: aggiungere più server. Funziona, e produce le architetture distribuite.

#### Primo approccio: replica

Un primo approccio per ottenere architettura distribuita è quello di ottenere la maggiore potenza attraverso le replicazioni (es. MySQL con replicate e memcached). Questo permette di ottenere availability.

Dietro le quinte si usa il transaction log; con il write-ahead log garantisce la persistenza, dato che posso sempre usare il log per disfare e rifare operazioni attraverso il log. Si usa quindi il log per creare una replica e mantenerla.

#### Log-based replication process



Tradizionalmente:

1. Scrivo prima sul log
2. Quando mi vengono richieste pagine dal buffer lo scarico e scrivo anche sul database
3. Il log viene letto da un replication process, che invia le operazioni da fare a uno slave DB il quale scrive sul replicated database.
4. Il replicated database scrive anche sul suo log.

Restano delle questioni: ho un DB replicato... ma è replicato completamente, non ho diviso il carico; sto solo rendendo il tutto più disponibile; **tutte le interrogazioni sono comunque svolte da un nodo solo**.

Non è male, ma se voglio fare di più mi serve distribuire davvero fra nodi.

### Tassonomia delle architetture distribuite

- **Shared-everything**: approccio vecchio, con un unico server dove il processo che gestisce i dati è su un solo nodo e tutte le risorse sono lì.
- **Shared-disk**: l'unica cosa in comune è la memoria secondaria, ma viene gestita da uno o più server che gestiscono. Tutto il resto invece viene eseguito sul suo nodo, con la sua CPU e memoria centrale.
  - Se ho delle cache è problematico gestire. La memoria secondaria resta il collo di bottiglia, e dunque viene abbandonato.
- **Share-nothing**: non condivide niente; ogni nodo ha il suo pezzo di dati e le sue risorse. Ognuno ha *un pezzo del db*, ovvero si applica lo sharding.
  - Ha tutta una serie di problemi sulla garanzia delle transazioni acide, e lo sharding diventa rilevante.

Analizzeremo qualche DB che usa lo standard share-nothing approach.

#### Architetture

- **Sharding tradizionale**: i dati vengono segmentati fra i nodi secondo una shard-key

- **Hadoop/HBase**: un processo globale determina dove dovrebbero andare le informazioni
- **Amazon dynamo consistent**: la allocazioni sui nodi è data da una funzione di hashing.

[Questa è una delle parti che chiede allo scritto!! NO ESERCIZI TH0] - 09/05/2022

## Architetture distribuite nei database visti

Bisogna trattare come questi sistemi gestiscono la replicazione del dato – o meglio, come lo distribuiscono.

La distribuzione è sempre orizzontale. È evidente che essendoci il problema del distribuire il dato sui nodi diventa rilevante sapere dove sono i nodi: il cluster potrebbe cambiare durante l'evoluzione del sistema.

Di conseguenza ogni sistema propone un suo modo.

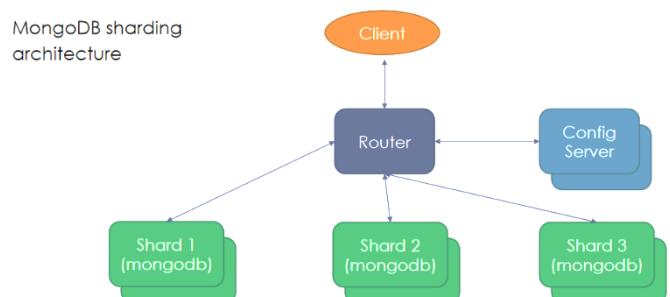
### MongoDB

#### Architettura

MongoDB alloca i dati su un nodo, e dopo di questo ciascun nodo lavora come se fosse un database di mongoDB separato. È una replicazione di buona parte del sistema su ciascun nodo.

L'applicazione che vuole interagire col D deve sapere dove andare a leggere e scrivere per ottenere i suoi dati.

- Il **config server**, un ulteriore database mongo, sa dove stanno i dati
- Il **router**, un nodo apposito, si occupa di ricevere le richieste, fare le interrogazioni al config server e reindirizzare le richieste.



#### Distribuzione

I dati sono distribuiti in vbuckets di distribuiti attraverso una **shard key**, che non abbiamo definito a livello logico ed è dietro le quinte. Possono esserci delle impostazioni di default ma sono modificabili; tipicamente è fatta da uno o più attributi indicizzati.

La distribuzione può essere di due tipi. Ciascun approccio ha vantaggi e svantaggi; il meccanismo di shard, in ogni caso, complica le interrogazioni e complica l'accesso ai dati.

<p>Range query: db.orders.find({“date”: {\$gt: yesterday}})</p> <p>Insertions: db.order.insert({“date”: today})</p> <p>Inserts are distributed evenly across all shards!</p>	<p><b>Range-based sharding</b></p> <p>Immagino di scegliere un attributo indicizzato e immaginiamo di avere uno shard per ogni gruppo (es. uno per lettera).</p> <ul style="list-style-type: none"> <li>- È facile sbilanciare il sistema 😞 Esistono dei rimedi, ma è un po' inevitabile.</li> </ul>
<p>All shards are involved!</p>	<p><b>Hashing</b></p> <p>Parto da una chiave applicativa ma poi ci calcolo una funzione di hash che definisce dove andranno i documenti.</p> <ul style="list-style-type: none"> <li>+ Ottengo che documenti con la stessa data vanno insieme, ma quelli con date vicine no. Quindi se poi voglio una interrogazione con cose “di fila” mi tocca interrogare tutti gli shard.; appesantisco, dato che ok vado in parallelo ma il router deve unire le cose.</li> <li>- Non ho più il problema dello sbilanciamento.</li> </ul>

! All'orale vuole anche i punti di debolezza 😊

### Bilanciamento

Il problema che si può generare è che ho lo sbilanciamento. Servirà fare un intervento periodico, che chiaramente rallenta il sistema. Tipicamente si tenta di evitare il grosso problema di riallocare tutto da 0 con un approccio **"shard chunk": suddivide gli shard in porzioni** di documenti ordinati fra loro secondo la chiave/hash e **rialloca questi pezzettini sui nodi fisici** senza intervenire più di tanto sull'allocazione vera e propria. È tutto automatico.

Nella metainformazione dovrò anche mantenere le info sugli shard chunk, ma mi semplifico la vita 😊

### !! Shard e così fisici potrebbero non coincidere. !!

### Replicazione

! Non è detto che ci sia di default. Potrebbe anche non esserci. !

Il meccanismo è quello del **replica set**: definisce degli insiemi di nodi:

- **Il nodo primario** è quello che gestisce la replica
- **I nodi secondari** accolgono le repliche e eventualmente aiutano a soddisfare le richieste che arrivano.

L'impostazione di default è che tutte le scritture arrivano al nodo primario, e sono propagate in modo asincrono sui secondari.

**Di sicuro i nodi secondari non sono sempre allineati col nodo primario! Le letture passano sempre dal primo.**

**Il nodo primario non è fisso, ma è definito per elezione:** se il primary node, che è quello che deve rispondere alle richieste che arrivano dal client, per caso va in failure gli altri se ne accorgono e lo sostituiscono. Per gestire questa cosa i nodi del replica set si scambiano sempre messaggi, e quando i messaggi non arrivano più capiscono che è morto. La condizione per rimanere primary è essere in grado di raggiungere almeno la metà dei nodi secondari.

La propagazione asincrona si basa sul **file di log**, chiamato qui **oplog**. I nodi secondari pescano dall'oplog e replicano le operazioni sulla loro versione dello shard che stanno gestendo. Oltre a questo meccanismo, i membri del replica set si scambiano dei **messaggi di heartbeat** che consentono di reagire a situazioni di guasto. Nelle elezioni ci sono meccanismi per evitare la parità.

Di **default** tutte le operazioni di scrittura vanno al primary e **l'operazione è conclusa quando sono concluse sul primary**; le altre sono asincrone.

Se il chiamante vuole invece rimanere bloccato fino a che tutte le copie hanno avuto l'aggiornamento si può fare una blocking call; è il chiamante a doverlo specificare e accettare il ritardo che ne conseguono. Addirittura può aspettare anche rispetto a un certo numero. Il client può chiedere, in caso di primary pieno, di accettare di leggere dal secondary node.

## HBase

“Se vi siete spaventati con MongoDB vi spaventerete il doppio con HBase. :D “

HBase ha un approccio del tutto diverso rispetto al relazionale. Si basa su HDFS, un file system che già di suo salva il blocco col file almeno 3 volte! Quindi il problema della replica è direttamente delegato a un livello più basso.

**L'approccio è completamente diverso rispetto al relazionale: si applica l'idea che ogni record non può mai essere modificato o cancellato.**

Per ogni modifica, HBase genera un nuovo file HDFS. Così, dunque, quando faccio le interrogazioni devo interrogare sia il database ben organizzato di base, sia quello “delta”. Nei momenti di tranquillità li mergo.

### Implementazione fisica

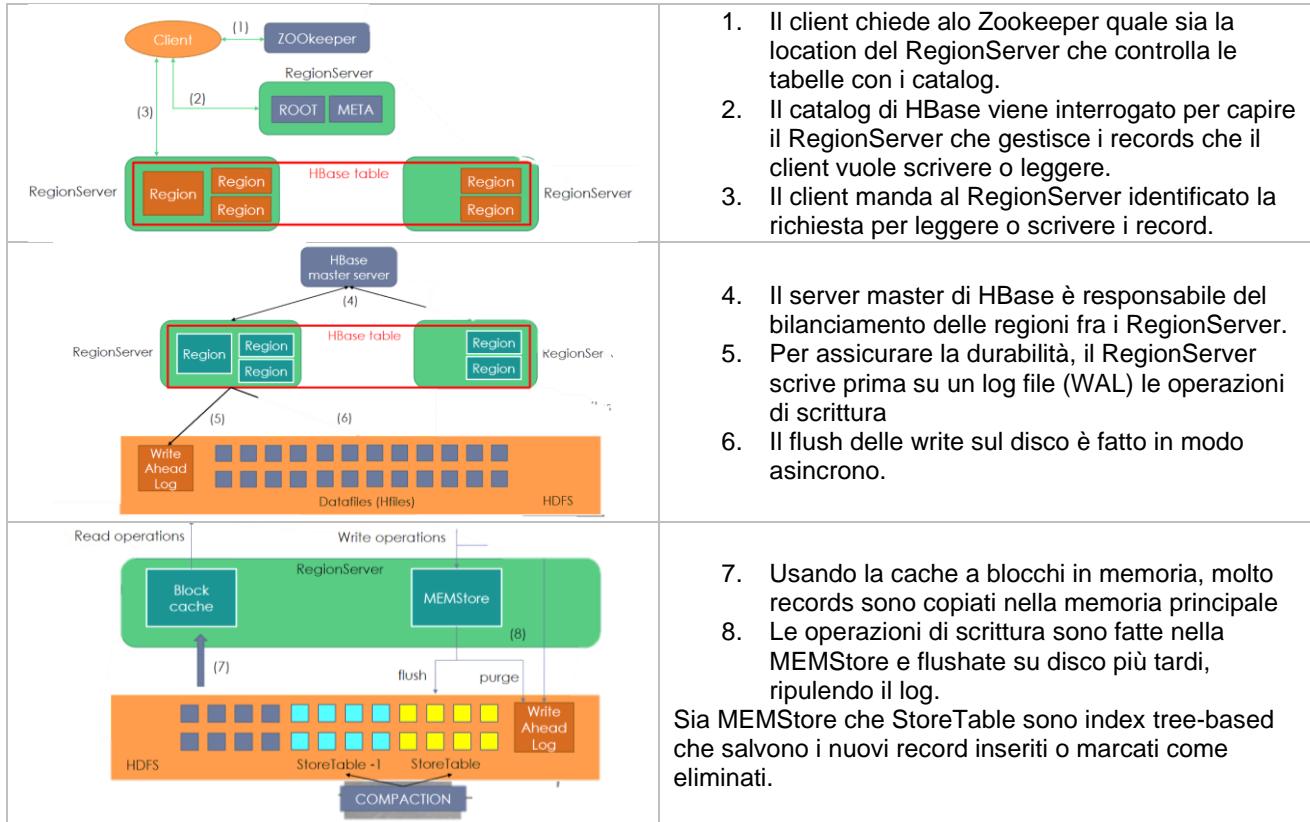
HBase è organizzato in tabelle di record con chiave e famiglia di colonne. Tutto questo finisce in files di tipo **Hfile**. Quando la tabella cresce molto, *come succede per tutte le tabelle HBase che si rispettino*, esse sono partizionate orizzontalmente in una o più **regions**. Ogni regione **assomiglia allo shard** e contiene **record con chiave contigua**.

Infine si istanziano dei processi per **gestire letture e scritture su queste regions**, detti **region server**. Tipicamente sono istanziate sullo stesso nodo della region, per principio di località.

- Dobbiamo sapere dove sono state memorizzate le region delle tabelle

- il client deve sapere a chi chiedere.

Lo **zookeeper** è un processo che sta dove si trova il **region server che contiene il catalogue**, ovvero la metainformazione.



- Anche qui abbiamo **problemi di bilanciamento** e quindi esistono **sistemi per gestire merge/split**.
- Abbiamo un **write ahead log**; quando il region server esegue un'operazione, la prima che fa è scrivere sul log che sta facendo quella scrittura; poi propagherà la scrittura in modo sincrono.
- Per le letture assomiglia al relazionale: si copiano in memoria dei blocchi della memoria secondaria e si tiene un **buffer** con il quale rispondere alle letture.
- Sulle scritture, invece, abbiamo un altro modo di operare: le **scritture sono fatte in una porzione "delta"** che ha un'idea simile a couchbase; in MEMstore sono inserite tutte le modifiche e inserimenti nel db. Ovviamente il modo di procedere segue la logica log->memstore->salvo memstore in memoria secondaria, generando diverse storetable, ovvero copie sul file system del memstore.

## Compaction

A un certo punto, periodicamente, **queste strutture saranno combinate fra loro**, compattate e inserite nel database vero e proprio. Dopo la compactation arriviamo di nuovo a uno stato di ordine.

Anche qui vengono introdotte le regionreplicas, perché è vero che non si perdevano i dati però avevo lo stesso un'interruzione del servizio finché si reinstantiava il server. Quindi ho più copie delle regioni in cui è suddivisa la tabella → copie delle copie. Nel momento in cui un regionserver fallisce, viene sostituito.

**Le scritture sulle repliche sono asincrone.** Non ho problemi, tranne nella situazione in cui ho un failure e non ho una copia aggiornata. Poco probabile.

## Dynamo (Cassandra)

È un sistema che “va di moda”.

L'approccio è ancora diverso; anziché avere alcuni processi con il compito di gestire la metainformazione e coordinare il lavoro, si decide che **tutti i nodi fanno tutto** all'occorrenza: i nodi assumono ruoli specifici nel momento in cui è necessario.

Per esempio, quando arriva una richiesta c'è un nodo che la gestisce direttamente, non è un certo nodo che si occupa di reindirizzare. Quando aggiungo un nuovo nodo, un altro nodo già esistente assume il ruolo di **seed node** (=gli passa le info per entrare a far parte della famiglia ❤️).

Per mantenere le info di come è organizzato, ogni nodo ha una sua copia della metainformazione (che potrebbe non essere alienata). I dati che indicano dove è memorizzata l'info e quali sono i nodi che lavorano/non funzionano sono distribuiti, e tutti ne hanno una copia. Queste info sono tenute aggiornate con un **Gossip protocol**.

### Gossip protocol

- È l'approccio del tutto opposto al relazionale, che invece tende a centralizzare.
- **Tollerò incosistenza temporanea anche nella metainformazione.**
- Ogni nodo manda agli altri una copia delle informazioni che lui ha riguardo il suo stato e alla metainformazione conosciuta. Chiaramente, se fallisce un nodo non perdo la metainformazione; invece prima se falliva il nodo master era un casino 🎰 In questo modo il **fallimento** di nodi avviene in maniera **probabilistica**
- → Il nodo che non aggiorna è **marcato come preoccupante**, e non gli vengono mandate richieste.

### Sharding

È basato sul **consistent hashing** partendo dalla **rowkey**. Si parte con un'hash function per dividere nei diversi nodi; tipicamente la hash function ha un range  $[2^{63} \dots 2^{63}-1]$ .

Cosa succede quando scalo, aggiungo nodi o devo ribilanciare il carico?

<p><i>Raddoppio il numero di nodi</i></p> <p>Una possibilità è quella di raddoppiare il numero di nodi, <b>dimezzando gli intervalli</b>. C'è da fare un minimo di ristrutturazione da fare, ma c'è anche una porzione di record che rimane dov'è 😊</p>	
<p><i>Aggiungo un solo nodo alla volta</i></p> <p>Posso procedere in due modi:</p> <ul style="list-style-type: none"> <li>• <b>Rimappo tutti i range di hashing</b>: molto costoso ma bilanciato</li> <li>• <b>Mappo il nuovo nodo in un range già esistente</b>: assegno metà dei ricordi di un nodo; metà li lascio lavorare come prima e l'altra metà la assegno a quello nuovo. Questo però produce situazioni sbilanciate, perché il nodo che ho rimappato ok ma tutti gli altri sono ancora pieni; dovrei intervenire anche sugli altri...</li> </ul>	
<p><i>Virtual nodes</i></p> <p>Il modo più bello è fare i virtual nodes, ovvero genero degli intervalli molto piccoli sul ring dei possibili valori della hash function e assegno a ciascun nodo vero un sottoinsieme di virtual nodes. Se un nodo diventa pesante, sposto dei VN da un nodo all'altro.</p>	

### Repliche

Anche qui posso gestire le repliche; le repliche sono repliche dei dati gestiti da un nodo – è uno “shard” che viene replicato un numero di volte che dipende dal parametro N – **tunable consistency**.

Di default, il coordinator node scrive copie in:

- **Simple replication strategy**: N-1 nodi successivi nell'ordine della chiave di hash, per avere un approccio semplice
- **Network topology aware replication strategy**: N-1 nodi diversi, scelti in modo tale che le copie siano su data center o server rack diversi.

## INTERNAL: CONSISTENCY MODELS [✓]

Parliamo della gestione della consistenza. È un problema ed è dei sistemi nuovi, quindi dobbiamo ridefinirla un attimo. Sappiamo già che nei relazionali questo problema non esiste, dato che il dato non è replicato e ha le proprietà acide.

Se invece passiamo alla nuova situazione bisogna capire il nuovo stato dell'arte. I nuovi sistemi abbandonano completamente l'implementazione delle transazioni e delle proprietà acide. Di conseguenza, si presenta il problema di gestire la consistenza in diversi ambiti; i sistemi nuovi forniscono un range di gestione della consistenza sempre riferito a un singolo oggetto. Questo vuol dire che in ogni caso non avrò mai un livello di consistenza uguale a quello di prima.

! La consistenza non potrà mai essere un requisito nei nuovi sistemi !

Inoltre, testare la consistenza è molto difficile perché dovrei generare il caso non solo nei suoi attori ma anche facendo interagire le cose nel momento X critico.

So che posso garantire diversi livelli di consistenza. In generale, devo capire come fare il tuning di questo aspetto dato che non è più garantito a priori.

Ci sono diversi modi di interpretare la consistenza:

- **Consistenza fra utenti:** a tutti gli utenti è data la stessa visione sul dato.
- **Consistenza in un a singola sessione (transazione):** se modifichiamo una riga in una sessione e leggiamo quella stessa riga vediamo la nostra update
- **Consistenza in una singola richiesta:** se accediamo a una tabella non vediamo le insert che accadono dopo che iniziamo la query. Voglio che nel momento in cui apro la DB non vedo le modifiche degli altri utenti fino a che io non ho concluso.
- **Consistenza con la realtà:** durante la sessione lo stato del database deve sempre riflettere la situazione reale del sistema informativo. Tipicamente si fa gestendo i tempi appropriatamente, cosa che non è implementata nei nuovi sistemi.

Vediamo le tecniche adottate per andare incontro a queste esigenze di consistenza.

### RDBMS consistency

#### Locking

Nei sistemi concorrenti, tipicamente, la tecnica è quella di **bloccare le risorse**; in particolare, il **locking a due fasi** e il **locking a due fasi stretto** garantiscono l'esecuzione safe delle transazioni. Questa roba è implementata in PostgreSQL, Oracle, etc. Il meccanismo di locking si basa sul fatto che **le risorse possono essere bloccate** – ovvero se una transazione vuole accedere a una determinata riga deve prima richiedere un lock in lettura/scrittura sulla riga; a seconda delle politiche messe in atto dal sistema, l'applicazione aspetta. Questo **garantisce la serializzabilità**.

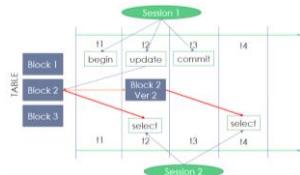
La gestione fatta in questo modo **rallenta molto**, quando abbiamo milioni di transazioni al secondo, e quindi non viene adottata.

#### MVCC

Anche sui sistemi relazionali ci sono modi di **attenuare** l'impatto di questo controllo. Un esempio è il MVCC, che evita di bloccare chi legge (=chi legge può sempre leggere).

L'idea è che in questa tecnica vengono mantenute più copie del dato marcate con un timestamp. Questo mi consente di costruire situazioni e stati di basi di dati nel passato che sono consistenti con ciò che è successo in quel momento; quando una transazione vuole accedere alla risorsa, l'accesso viene dato allo snapshot più recente che precede il timestamp della transazione. Anziché i timestamp posso anche usare i System Change Number. --> **abbandono la consistency for session**.

Esempio:



Il dato sarebbe bloccato e la sessione 2 non potrebbe accedere (dato che la sessione 1 lo sta modificando). Invece può accedere alla versione precedente.

Quesione: atomicità delle transazioni in RDBMS distribuiti

Nel caso in cui la mia transazione coinvolga più shard, come garantisco di fare il commit su tutti i nodi? i hanno già pensato, con il **two-phases-commit protocol (2PC)**. 😊 Anche questo però rallenta, perché implica che tutti si devono fermare per scambiarsi messaggi 😞

2PC – two phases commit protocol

L'idea è che ci sia un coordinatore, il cosiddetto transaction manager (TM), che è il nodo da cui è partita la transazione, e tutti gli altri nodi coinvolti, i resource managers (RM).

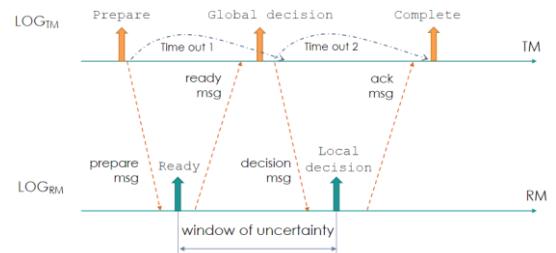
Si basa sul logo, e introduco **nuovi record di log**, come prepare, global commit, complete, ready. Questi record sono scritti in vari log durante l'esecuzione del protocollo di commit a 2 fa

- **Prepare record:** scritto dal TM a inizio transazione.
- **Global commit/abort record:** scritto dal TM quando è deciso il risultato della transazione.
- **Complete record:** Scritto dal TM quando tutti hanno ricevuto il commit, ovvero quando il protocollo è terminato e i lock sono rilasciati
- **Ready record:** Scritto dai RM quando sono pronti a partecipare al protocollo.

Se ci sono problemi – ad esempio il messaggio di ready non arriva entro un certo tempo – si decide per un global rollback.

Se viene perso un messaggio di ack si rimanda il messaggio con la decisione presa finché il RM non risponde di essere riuscito.

**Nzomma tutte ste cose allungano ancora di più la latenza, quindi non è accettabile.**



## Riduzioni di consistenza

Consistenza mono oggetto

Non possiamo gestire una consistenza che gestisce più oggetto nello stesso momento.

Questo è importante dei DBMS relazionali più che nei nuovi, perché spesso facendo normalizzazione e frammentando le info su più tabelle anche **per una sola modifica è necessario gestire tante tabelle**, e quindi bisogna garantire la consistenza fra esse.

Nei sistemi nuovi invece si tende a **incapsulare** l'informazione, e quindi poi fisicamente quando blocco il documento blocco tutta l'info che mi servirà; se così non è vuol dire che ho fatto male la modellazione. 😊 😊 😊

## Livelli di consistenza sulla singola operazione

- **Consistenza stretta:** una read ritorna sempre il valore più recente.
- **Eventual consistency:** già vista, il sistema potrebbe essere inconsistente in un certo punto, ma tutte le operazioni individuali prima o poi saranno applicate consistentemente. Se tutte le update si fermano, il sistema arriverà a uno stato consistente.
  - **Read your own writes:** è un'*eventual consistency* dove si garantisce di vedere almeno le proprie operazioni
- **Weak consistency:** il sistema non garantisce di diventare mai consistente. Per esempio, se alcuni nodi falliscono l'update potrebbe andare persa.

## Consistenza in MongoDB

È il sistema **più vicino al relazionale**, e dunque cerca di garantire la **consistenza maggiore possibile**. Sin dall'inizio, garantisce la **consistenza stretta sul singolo documento**; il documento è gestito in modo atomico e gli utenti lo vedranno sempre in un solo stato.

Per garantire ciò, MongoDB applica il **blocco ai documenti che sono in modifica** e non implementa nessuna versione di MVCC, ovvero **non c'è un rilascio del lock per le letture**: se una roba è bloccata non la legge nessuno.

All'inizio questo aveva una **granularità inesistente** (TUTTO il DB), poi per fortuna pian piano la granularità è passata a istanze di info più piccole fino ad arrivare a un **lock sui singoli documenti** nelle versioni attuali.

I problemi di consistenza vengono principalmente dai **replica set**; se ci sono, la consistenza dipende tutta da come le ho impostate:

- Se le **lettura passano sempre dal primary node** ho ancora la consistency
- Se voglio accedere alle copie, aggiornate in asincrono, potrei non avere consistenza forte ma solo **eventual consistency**.

Non c'è molto da dire: MongoDB è quello che più si avvicina alla consistenza relazionale. Ergo, è il più lento.



## Consistenza in HBase

HBase è *un po' meno bravino (cit)* sulla consistenza; l'idea è di **bloccare la singola istanza di informazione (riga)** della tabella, con tutte le sue column family. Il **region server** è il processo che gestisce questa cosa.

**Le letture, però, non sono bloccate dalle scritture.** Applico **MVCC**: le le scritture/lettura sono concorrenti, la lettura legge la versione precedente. L'accesso si baserà sul **System Change Number**:

- Se eseguo scrittura, prendo il numero corrente di scrittura (**WN**) e lo memorizzo insieme alla riga che sto modificando; marco la riga.
- Quanto inizia la lettura, si assegna un numero di lettura **RPN** con il quale identifica la scrittura più recente che può leggere avendo iniziato a leggere in quel momento; in pratica può leggere solo ai **WN < RPN**.

Fin qua siamo nella totale consistenza. Ma se abbiamo le region replica, **se accedo alle copie posso avere problemi di eventual consistency**. Posso settare questa cosa dicendo che la consistenza della lettura è settata alla **timeline consistency**, ovvero seguo la linea del tempo per far accedere le letture alla corretta visione della DB. Ma se ho l'**eventual consistency**, siccome una volta potrei leggere dal primario e una volta dal secondario, rischio di non accedere al dato modificato.

## Cassandra

*Cassandra, come dice il nome, è quello che fa il disastro totale sulla consistenza.* 🤪🤪🤪 È quello che mette su più trucchi strani, ma alla fine non è mai consistente, o meglio di default **ammette l'inconsistenza di default**.

Questo perché qui ho la **tunable consistency**: posso definire la consistenza come la voglio io, e mettere dei paletti per renderla meno probabile.

Posso tarare la tunable consistency attraverso tre parametri:

- **Replication factor**: numero di copie dei dati, solitamente è 3.
- **Write consistency level**: quante copie devo scrivere ed essere sicuro di avere scritto prima di restituire il "fatto!" a chi mi chiede. Più è alto, più faccio aspettare.
- **Read consistency level**: quante copie devo leggere prima di decidere quale valore tornare. Questo perché potrei leggere valori diversi; se riesco a leggerle tutte, di sicuro almeno una è quella giusta 😊

### Livelli di consistenza (sulla write)

- **ALL**: la write deve arrivare a tutti i nodi. Di fatto sto avendo una strict consistency.
- **ONE | TWO | THREE**: propagata a quel numero di nodi
- **QUORUM**: la maggioranza 😊
- **EACH\_QUORUM**: la maggioranza in ciascun data center
- **LOCAL\_QUORUM**: la maggioranza in *questo* data center
- **ANY**: scrivo una sola modifica, e potrei persino scriverla su un nodo che *non* ha quel dato; qualunque nodo io raggiunga, il marco lì che andrà fatta la modifica (hinted handoff). Questa nota rimane lì per un certo lasso di tempo che decido. Il nodo che si prende in carico questa roba tenterà di aggiornare; se scade il timeout la modifica è persa.

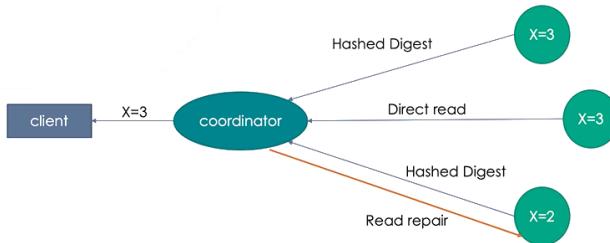
Quelli della read sono simili.

WRITE \ READ	ONE	QUORUM	ALL
ONE	High performance and availability, but no consistency		Fast and high available for writes; consistent but slow reads
QUORUM		Intermediate performance; good availability; strong consistency	
ALL	Slow and not highly available for write; fast and consistent reads		Strictest consistency, but lowest availability and performance

### Handoff

È la situazione nella quale ho essere il depositario del dato ma che non riesco a contattare; dunque, scrivo su un altro nodo una nota con la verifica da fare. Entro un certo tempo, quando l'altro nodo diventerà raggiungibile il secondo nodo si occuperà di fare la modifica; else la modifica si perde.

Read repair: sistemare

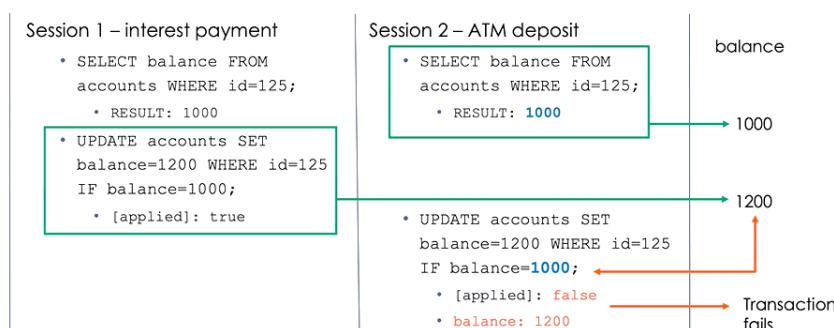


Quando faccio una lettura, in realtà leggo N nodi. Anziché leggere tutto tutto leggo una rappresentazione compatta (hash) e verifico se ci sono letture diverse. Quando ottengo un dato diverso (e lo vedo anche solo dall'hash), uso il timestamp per sapere quale sia quello corretto: ritorno il valore corretto al lettore e poi riallineo il DB.

### Transazion lightweight

Cassandra tenta di inserire dei meccanismi che consentano di inserire meccanismi di transazione concorrente inglobando assieme atomicamente scritture e letture; ovvero, se voglio modificare un dato ed essere certa che nessuno stia modificando quel dato insieme a me allora lo leggo e mi salvo la copia del dato che voglio modificare; infine dico al sistema di modificarlo solo se è rimasto come l'ho letto; altrimenti qualcun altro deve averlo modificato.

Non è proprio una transazione; è solo il fatto che non voglio perdere degli aggiornamenti (es. "moltiplica per due quello che c'è" -> qualcuno si frappone fra scrittura e lettura).



Inoltre, ho anche il problema delle repliche: in questo caso sarà necessario introdurre qualcosa di simile al lock a due fasi, perché è necessario che tutti si mettano d'accordo per ogni aggiornamento da fare.

Qui si ragiona per maggioranza: se la maggioranza delle repliche dice ok allora si va avanti, anche se le altre dicono di no e quindi di nuovo rischio di generare inconsistentezze. Però, asu usual, se ho che leggo dalla maggioranza poi questo problema si risolve.

Ha finito di parlare male degli altri sistemi. Cit.

## DATABASE SPAZIALI: FONDAMENTO TEORICO [✓]

Sono database riferiti a **questioni geografiche-spatiali**. Nel passato questi sistemi erano dedicati proprio al dato geografico, inteso come fenomeni che accadono sulla superficie terrestre; la nostra descrizione di ciò che accade sulla superficie terrestre ha subito un'evoluzione tecnologica molto più di altri ambiti.

- Tradizionalmente, la rappresentazione avveniva con le **carte geografiche** – che era la DB del passato – e i suoi limiti influenzavano sia la **fruizione** che la **memorizzazione**.
- Da qui si passa a rappresentare ciò che c'era sulla carta dentro il calcolatore, con una rappresentazione che descrive il modo di disegnare gli elementi con una cartografia digitale i cui obiettivo non era ancora di interrogare direttamente le carte, ma **generare le carte (fisiche) su cui lavorare**.
- Infine, si passa finalmente a far coincidere i due mondi: si permette di **fruire del dato direttamente visualizzandolo sul calcolatore**, con tutta una serie di problemi (per esempio, la precisione delle misurazioni può essere molto minore per la carta dato che non ho zoom in). Inoltre, essendo misurazioni molto costose, spesso arrivano “a pezzi” nel corso degli anni :’)

La filiera, un tempo, era quella dei voli aerei con macchine stereoscopiche; si restituiva il vettoriale partendo dai **fotogrammi**. Fortunatamente, con gli anni, sono arrivate **fonti gratuite** tipo OSM e Google; non sempre, però, sono le informazioni precise (tubi? Precisione al millimetro?) di cui ho bisogno.

Aspetto problematico: lavorando a diversi livelli cambia il modo di rappresentare l'informazione e le problematiche.

Un tempo la cartografia era indipendente: avendo molti usi diversi, la si faceva in un certo modo e si usava sempre quella. Al giorno d'oggi, la rappresentazione dipende anche dall'applicazione. L'informatica ha cambiato il modo di lavorare.

- Le interfacce sono solitamente **grafiche**
- Ci sono **tanti sistemi di riferimento possibili**; tutti, comunque, deformano e quindi hanno delle tarature caratterizzate diversamente. Prima con le carte non avevo questo problema: 1 carta 1 SDR. Ora invece potrei avere i dati con SDR diversi.

L.18 - 16/05/2022

Come rappresento le informazioni fisicamente sul calcolatore?

### Formato raster

È una **griglia**, e i dati sono associati a **una cella della griglia**. La descrizione è molto compatta e generalizzata.  
È facile interagirci come essere umano, meno comodo interrogarla stile base di dati.

### Formato vettoriale

Lo spazio è rappresentato come **insieme di oggetti che voglio collocare** sul territorio. La scena si costruisce popolando lo spazio con gli oggetti; chiaramente devo poter descrivere l'oggetto nello spazio, usando punti, curve e superfici triangolate, solidi, coordinate ed equazioni per indicare posizione ed estensione

Ad un livello puramente concettuale voglio solo **descrivere le proprietà astratte delle geometrie**, senza indicare il modo in cui le rappresento. Per esempio, in analisi, non rappresento mai graficamente una retta (=elenco di coppie di punti) ma lo faccio sempre in maniera astratta (=equazione).

## Rappresentazione vettoriale

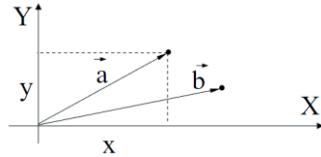
La rappresentazione vettoriale si basa sulla **geometria dei vettori**. Lo spazio di partenza è il piano 2D. In questo mondo ci si è mossi assegnando dei modelli astratti per dire quali siano le geometrie da rappresentare in un certo modello + una rappresentazione fisica effettiva. Quindi non ho una rappresentazione intuitiva, non è un'equazione ma un insieme di triangoli o rettangoli con un loro studio teorico. (non ho capito questa intro quindi la ignorerò, bacioni)

### Punto

L'elemento base è il **punto, identificato da un vettore**. Copre buona parte delle informazioni necessarie. La rappresentazione vettoriale si basa sulla **geometria dei vettori**, per cui partiamo da uno spazio di riferimento che tipicamente è un piano euclideo. La distanza è definita come la norma del vettore differenza, o teorema di Pitagora.

Operazioni sui vettori:

- $\vec{a} + \vec{b} = \vec{c}$  con  $\vec{c} = (x_1 + x_b, y_a + y_b)$
- $\vec{a} - \vec{b} = \vec{d}$  con  $\vec{d} = (x_1 - x_b, y_a - y_b)$
- $\lambda\vec{a} = \vec{e}$  con  $\vec{e} = (\lambda x_a, \lambda y_a)$



## Segmento

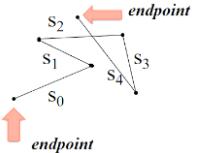
È rappresentato in maniera astratta come **tutti i punti ottenibili da questa formula**:

$$\vec{s} = \{\lambda\vec{a} + (1 - \lambda)\vec{b} \mid \lambda \in [0,1]\}$$

Per rappresentarlo, ovviamente, non vado a rappresentare questi infiniti punti ma **memorizzo le caratteristiche** necessarie per ottenerlo (es. le coordinate di A e B).

## Polyline

Le prime architetture non potevano disegnare curve quindi usavano una **sequenza di N segmenti** anche per le curve. Metto i segmenti in sequenza in modo che dove termina uno inizia l'altro. Ogni endpoint è condiviso al massimo da due segmenti:  $sp = (s_0 \dots s_{n-1})$



Si introducono alcune proprietà:

• Una polyline è <b>semplice</b> quando non ha intersezioni.	$(\forall s_i, s_k \in sp)(i \neq j \Rightarrow (s_i \cap s_j = \emptyset \vee ((j = (i+1)_{mod n}) \wedge (s_i.E_0 = s_j.E_0 \vee s_i.E_1 \vee s_i.E_1 = s_j.E_0 \vee s_i.E_1 = s_j.E_1)))$
• Una polyline è <b>chiusa</b> o ciclica se l'ultimo estremo coincide con quello di partenza.	$s_0.E_0 = s_{n-1}.E_1$
• Una polyline può essere <b>orientata</b> quando stabiliamo quale sia il primo e l'ultimo segmento.	<p>polyline      simple polyline      simple and cycle polyline non simple and cycle polyline</p>

## Polygoni

Un poligono è una **porzione di spazio definita da una polyline semplice e chiusa**. Un teorema della topologia assicura che una polyline chiuse e semplice identifica sempre una parte **finita** di spazio.

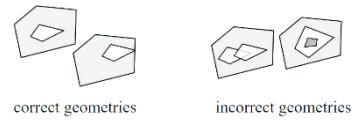
In generale, potremmo anche definire il poligono come insieme aperto – ovvero la frontiera non è inclusa. Nella realtà tutti gli oggetti sono limitati, quindi in realtà di fatto definiamo gli oggetti come chiusi in senso topologico – ovvero includono la frontiera. Altre proprietà:

### • Poligono con buchi:

Il poligono con buchi è definito da una linea poligonale semplice e chiusa, che definisce il poligono senza buchi, e tutte le altre definiscono eventuali buchi.

Ci servono ad esempio, per rappresentare edifici con cortiletto dentro.

- Tutte le linee dei buchi devono essere contenute nel poligono esterno
- Non si devono intersecare tra di loro perché non saprei definirne la semanticità
- Ammessa solo l'intersezione in un punto.



$\forall (sp_i, sp_j \in H)(i \neq j \Rightarrow ((sp_i \text{ does not intersect } sp_j) \vee (sp_i \text{ intersects } sp_j \text{ only in one point})) \wedge (sp_i \text{ is not contained in the polygon delimited by } sp_j))$

## Relazioni spaziali

Rappresentare le relazioni in un DB spaziale è un bordello, perché molto dipende dalla rappresentazione (es. la stessa cosa la posso rappresentare in 1000 volte, e l'utente graficamente non se ne rende conto). Dunque, interrogare è sempre complicato e **stabilire relazioni è complicato**.

Le relazioni che vogliamo considerare noi sono **binarie**, ed esistono o non esistono fra due oggetti. Esistono diversi tipi di relazione; le principali sono le **topologiche**. Le altre relazioni sono meno enfatizzate, in quanto un po' figlie di quelle topologiche; dal punto di vista dell'utente finale, comunque, sono solo modi diversi di relazionare le cose.

## Relazioni topologiche

### Definizione

Una relazione  $R$  è detta **topologica** se è **invariante rispetto a una deformazione topologica dello spazio** – ovvero ai cosiddetti rubber sheet transformation, trasformazioni che stirano lo spazio come fose un foglio di gomma senza generare strappi o pieghe.

Poprietà:

- **Indipendenti dalla distanza**, quindi sarebbero vere anche in uno spazio non euclideo
- **Qualitative**: descrivono come due geometrie sono in relazione senza specificare misure
- **Si riferiscono a concetti di alto livello**, e spesso hanno una parola corrispondente in linguaggio naturale.

### Modello di Egenhofer

La definizione formale è data dal modello di Egenhofer. Questo modello si basa su una certa rappresentazione degli oggetti, definita in maniera astratta. Il modello, dunque, **vede gli oggetti come insiemi di punti**.

Gli oggetti devono essere fatti in modo che sia definito un **partizionamento dello spazio rispetto all'oggetto  $\lambda$** :

- **Interno** di  $\lambda \rightarrow \lambda^\circ$
- **Frontiera/boundary** di  $\lambda \rightarrow \partial\lambda$
- **Esterno** di  $\lambda \rightarrow \lambda^-$

Il vero problema è **definire il boundary**, perché non ne esiste una definizione univoca in matematica. La definizione **topologica** di boundary rende il boundary stesso **dipendente dallo spazio dove l'oggetto è immerso**. Se ho una linea in uno spazio 3D, la definizione mi dice che la linea è boundary. Ma se invece sono in uno spazio 2D mi trovo che il boundary sono i due punti al limite, poiché cambia la definizione di intorno. Dunque mi serve una definizione di boundary indipendente, e viene fatto in **matematica** con i **boundary combinatoriali**.

Il risultato è, nello spazio 2D o 3D:

**Punti**: empty boundary

**Curve**: combinatoriale (punti ai bordi)

**Surface**: combinatoriale. (polyline)

Dato che tutti gli oggetti hanno queste tre parti, posso **definire le relazioni intersecando queste tre parti**; così **definisco tutte le possibili relazioni**.

Una relazione si definisce dicendo che una di queste configurazione è **non vuota** e tutte le altre sono vuote.

Non tutte sono possibili: scarto tutte le relazioni che prevedono che non ci siano elementi nell'intersezione con lo spazio esterno, dato che ho oggetti finiti. Non per tutte queste configurazioni esiste uno scenario che le soddisfa: dipende dalla tipologia di spazio in cui sono immerso e dalla tipologia di oggetto.

Alla fine, ci sono **512 relazioni vere** che si possono definire su casi reali; il modello poi si applica a tutti i casi significativi. In particolare, prendendo due poligoni nel piano, riesco a definirle tutte.

Esempio:

		$B^\circ$	$\partial B$	$B^-$
		$A^\circ$	$\emptyset$	$\emptyset$
		$\partial A$	$\emptyset$	$\emptyset$
	$A^\circ$	$\emptyset$	$\emptyset$	$\emptyset$
	$\partial A$	$\emptyset$	$\emptyset$	$\emptyset$
	$A^-$	$\emptyset$	$\emptyset$	$\emptyset$

$B^\circ$	$\partial B$	$B^-$	
$A^\circ$	$A^\circ \cap B^\circ$	$A^\circ \cap \partial B$	$A^\circ \cap B^-$
$\partial A$	$\partial A \cap B^\circ$	$\partial A \cap \partial B$	$\partial A \cap B^-$
$A^-$	$A^- \cap B^\circ$	$A^- \cap \partial B$	$A^- \cap B^-$

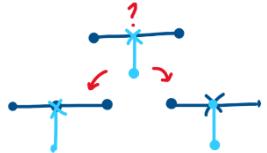
### Relazioni di Clementini

Introdurre 512 nomi è un po' impossibile, quindi si propone un modo diverso di definire le relazioni; si definiscono delle relazioni più intuitive per poi dare un linguaggio che permette di rifinirle. Queste sono le relazioni di Clementini. Ci si basa sul fatto che so sempre definire spazio di frontiera, interno e esterno.

Nome	Definizione	Spiegazione	Disegnetti poligoni	Disegnetti linee	Disegnetti linee+poligoni
<b>DISJOINT</b>	$\lambda_1 \text{DISJOINT} \lambda_2 \Leftrightarrow (\lambda_1 \cap \lambda_2 = \emptyset)$	I due oggetti non hanno punto in comune.			
<b>TOUCH</b>	$\lambda_1 \text{TOUCH} \lambda_2 \Leftrightarrow (\lambda_1^\circ \cap \lambda_2^\circ = \emptyset) \wedge (\lambda_1 \cap \lambda_2 \neq \emptyset)$	Le parti interne non si toccano ma gli oggetti hanno delle parti in comune.			 (basta non si tocchino dentro...)
<b>OVERLAP</b>	$\lambda_1 \text{OVERLAP} \lambda_2 \Leftrightarrow \dim(\lambda_1^\circ \cap \lambda_2^\circ) = \dim(\lambda_1) = \dim(\lambda_2) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$	Definisco l'overlap fra due oggetti della stessa dimensionalità. Le parti interne si devono intersecare, e la dimensionalità dell'intersezione deve essere uguale alla dimensionalità degli oggetti; ad esempio, se ho due poligoni l'intersezione deve avere dimensionalità due. Devo anche essere certa di non avere un contenimento.		 (condivide linea, che ha dim = 1)	Nope!
<b>CROSS</b>	$\lambda_1 \text{CROSS} \lambda_2 \Leftrightarrow \dim(\lambda_1^\circ \cap \lambda_2^\circ) \leq (\max(\dim(\lambda_1), \dim(\lambda_2))) - 1 \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$	È come l'overlap, ma la dimensionalità dell'intersezione è inferiore di uno della dimensionalità degli oggetti che partecipano alla relazione. Ad esempio, se ho due linee l'intersezione è un punto.		 (condivide punto, che ha dim = 0)	
<b>IN</b>	$\lambda_1 \text{IN} \lambda_2 \Leftrightarrow (\lambda_1^\circ \cap \lambda_2^\circ \neq \emptyset) \wedge (\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$	L'intersezione tra gli interior non è vuota. E l'intersezione degli oggetti deve darmi uno dei due (non il viceversa, altrimenti sarebbe uguale)			
<b>CONTAINS</b>	$\lambda_1 \text{CONTAINS} \lambda_2 \Leftrightarrow \lambda_2 \text{ IN } \lambda_1$	È il viceversa.			
<b>EQUAL</b>	$\lambda_1 \text{EQUAL} \lambda_2 \Leftrightarrow (\lambda_1^\circ \cap \lambda_2^\circ \neq \emptyset) \wedge (\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1 \cap \lambda_2 = \lambda_2)$	È ovvio ig			?????

### Note:

- Non posso fare un CROSS fra due poligoni perché sto parlando di intersezione della parte interna. Si può nel 3D: basta metterli perpendicolarmente nello spazio.
- Con i poligoni va circa tutto bene, perché i poligoni hanno la stessa dimensionalità dello spazio, quindi non ho stranezze. Se prendo le linee le cose si complicano: avendo dimensionalità inferiore fanno qualche scherzo.
- Gestire il **touch nelle linee** è parecchio complesso:
  - Innanzitutto, la rappresentazione delle curve in realtà avviene attraverso **spezzate**.
  - **Con coordinate finite, andare esattamente a beccare il punto preciso che sta sulla curva è quasi 0**. Questo significa che nei sistemi veri andare è quasi impossibile beccare un vero TOUCH a true! Quindi in realtà dovrò stabilire una certa **tolleranza**.  
Per essere sicuro che si intersechino, posso anche rappresentare mettendo apposta dei punti dove si intersecano le cose di modo da far condividere lo stesso punto (es. strutture fatte in modo che nella struttura ho proprio lo stesso punto)



**DJ:**  $a.DJ(b) \triangleq a.PS() \cap b.PS() = \emptyset$

$$R_{dj}(pt, pt) = [FFT\ FFF\ TFT]$$

$$R_{dj}(pt, c/s) = [FFT\ FFF\ T * T]$$

$$R_{dj}(c/s, pt) = R_{dj}(pt, c/s)^T$$

$$R_{dj}(c/s, c/s) = [FFT\ FF * T * T]$$

**Osservazione:** ogni relazione di clementini corrisponde a una o più matrici di Egenhofer. F sta per intersezione vuota, T per non vuota. Asterisco significa che non importa. Einhofen è più dettagliato, permette di distinguere quasi tutti i sottocasi.

L. 19 – 17/05/2022

### Relazioni direction-based

**Si basa sulle direzioni.** Nello spazio tradizionale euclideo non ci sono direzioni predefinite; è necessario introdurre un modello che rappresenti le direzioni prese in riferimento, che tipicamente sono i punti cardinali N/S/E/W. La rappresentazione di questo concetto è un partizionamento dello spazio dato un oggetto, che permette di stabilire quali punti si trovano a nord/sud/est/ovest. Questo può essere fatto in modi diversi:

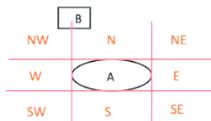
#### Privilegiamo esattamente nord sud est ovest:

Scelgo un punto di riferimento, vi faccio cadere le 4 tiles e poi vedo dove cade il secondo punto. Finché sono punti tutto ok; se invece abbiamo punti con estensione diversa da 0 diventa un po' più complicata.



#### Orientiamo il partizionamento direttamente sugli assi N/S/E/W:

Quindi le tiles risulteranno sfasate su NE/NW/SE/SW.



Se anziché un punto ho un oggetto con un'estensione, posso tenerne conto. Ma in questo caso, per esempio, mi esplodono i casi possibili; non tutte le configurazioni sono possibili, ma ottengo 169 possibili relazioni...

Pertanto, la cosa non è stata implementata perfettamente ma posso generare i tiles e calcolare la relazione attraverso questi. Non sono molto comuni.

### Relazioni distance-based

Il nostro spazio di riferimento è, per definizione, euclideo; quindi esiste una relazione di distanza fra punti

$$D(P, Q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Occorre estendere questa distanza esistente anche agli oggetti con dimensione diversa da 0; lo faccio definendola come il minimo fra tutte le distanze possibili fra i punti appartenenti a ciascuno dei due oggetti.

$$Dist(A, B) = \min (\{D(P, Q) : P \in A, Q \in B\})$$

Posso definire una relazione definendo una distanza minima e massima tale per cui i due oggetti sono in relazione se e solo se hanno distanza compresa fra i due valori fissati.

$$R_{(d_1, d_2)}(A, B) \Leftrightarrow (d_1 < Dist(A, B) < d_2)$$

## DATABASE SPAZIALI: GEO-RDBMS [✓]

Anche questi sistemi hanno avuto la loro storia: la sigla GIS è nota ma è anche abbastanza arretrata. Il primo obiettivo era di produrre **cartografia**, più che di interrogare la base di dati. Prima c'era persino **hardware dedicato**, poi evoluto sempre più verso i DB con Oracle e Postgres che integrano questo dato. **Ci si ferma al passaggio col relazionale**; nei sistemi nuovi per ora il dato spaziale è entrato (es. c'è in MongoDB) ma il supporto è limitato.

I dati spaziali esistono sotto tre forme:

<b>Relazionali estesi</b> (postgres, oracle...)	<b>Sistemi dedicati a applicazioni specifici</b> AutoCAD gestisce dato geografico con il file DXF, o sistemi ESRI (compagnia monopolista) con lo shapefile+DBF (geometrie + tabella di info associate)	<b>Dato semistrutturato</b> Esiste il file GML, un aiuto del XML, ma che ormai è super pesante e quindi al solito si è passati a GeoJSON.
---	---	--

Nel relazionale:

### DBMS relazionale tradizionale

È evidente che si poteva immaginare di costruire un insieme di tabelle senza inventare nuovi domini; usavamo int per le coordinate e costruire **ambaradom**. Questo insieme di tabelle, però, era molto complesso da utilizzare e l'indicizzazione non era per nulla adatta a gestire questo tipo di interrogazione. L'interfaccia grafica è irrinunciabile; tabelle di coordinate sono illeggibili...

### Approccio object oriented

C'è stata qualche proposta ma non ha funzionato. Erano molto robuste ma poco efficienti; inoltre, l'integrazione con applicazioni che rappresentavano ancora il modello relazionale era troppo difficile

Quindi, la soluzione sono i **Geo-DBMS**, estensioni dei RDBMS tradizionali detti geo-relazionali. All'inizio c'è stato il bisogno di definire precisamente cosa si intendesse per geo-cose: le caratteristiche di un **geoco** sono

- **Essere relazionali**, con la gestione della **transazione**
- **Implementazione di tipi specifici** per il dato spaziale
- **Estensione del linguaggio di interrogazione** per la componente spaziale
- **Implementazione specifica del join basato su condizioni spaziali**: è una fra le operazioni più pesanti, ma è anche pesantissima, tanto che ci vogliono a volte ore per farle.

Di conseguenza, bisogna estendere **modello**, **linguaggio** di interrogazione e **indicizzazione**.

### Standard SFS

All'inizio ciascun sistema aveva un modello; adesso sia Oracle che Postgres implementano lo standard del OGC (oracle geospatial consortium), che è **“Simple Features Specifications for SQL”**. Lo standard definisce che un database già relazionale deve essere considerato come un insieme di **feature**, dove la feature/geo-object è un'istanza di informazione che ha una connotazione spaziale. Le altre informazioni sono rappresentate in altre colonne, che utilizzano i tipi standard dell'SQL2.

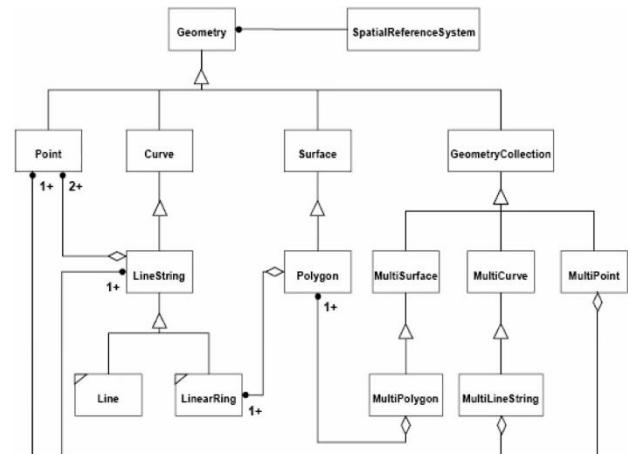
All'epoca c'era ancora la possibilità di immaginare di rappresentare nel relazionale tutti i contenuti attraverso colonne; poi c'era la versione che richiedeva la definizione di tipi specifici, che è quella che considereremo noi.

I costrutti sono:

- **Domini di base**: SQL92 + domini dati da SFS
  - **Relation/table**: tuple omogenee, as usual
  - **Indici spaziali**: sono strutture per accedere ai dati spaziali in base allo spazio di riferimento. Esistono R.tree, quad-tree, GiST,...
- Questi indici cercano di velocizzare le interrogazioni che riguardano geometrie in una certa porzione di riferimento di spazio.

### Geometry types (domini di base)

Note:



- Ci limitiamo allo spazio 2D.
- Gli oggetti che rappresentiamo sono sempre oggetti topologicamente chiusi (=includiamo la frontiera) e finiti.

Un database georelazionale - qui prendiamo l'esempio del GIS di Postgres - è composto da:

<b>Tabella che contiene tutti i sistemi di riferimento disponibili</b>	Sono molti perché ogni paese del mondo ha il suo. Contiene diversi attributi <ul style="list-style-type: none"> <li>• L'id è casuale ma è settato da uno standard internazionale</li> <li>• <b>SRTEXT: rappresentazione testuale di tutti i parametri del sistema di riferimento; anche lui non sa i dettagli.</b></li> </ul>	<pre>CREATE TABLE SPATIAL_REF_SYS (   SRID INTEGER NOT NULL PRIMARY KEY,   AUTH_NAME VARCHAR (256),   AUTH_SRID INTEGER,   SRTEXT VARCHAR (2048) )</pre>
<b>Tabella di metainformazione</b>	Spiega in quali colonne della base di dati ci sono valori spaziali. Consente ai sistemi che si interfacciano sapere dove pescare le info da visualizzare geograficamente. Specifica il nome della tabella, lo schema, il database, il nome della colonna, le dimensioni (2 o 3), il sistema di riferimento e un insieme di vincoli	<pre>CREATE TABLE GEOMETRY_COLUMNS (   F_TABLE_CATALOG VARCHAR(256) NOT NULL,   F_TABLE_SCHEMA VARCHAR(256) NOT NULL,   F_TABLE_NAME VARCHAR(256) NOT NULL,   F_GEOGRAPHY_COLUMN VARCHAR(256) NOT NULL,   COORD_DIMENSION INTEGER,   SRID INTEGER REFERENCES SPATIAL_REF_SYS,   CONSTRAINT GC_PK PRIMARY KEY (F_TABLE_CATALOG,   F_TABLE_SCHEMA,   F_TABLE_NAME, F_GEOGRAPHY_COLUMN) )</pre>
<b>Tipi geometrici spaziali</b>	Sono lo schema visto prima; solo un sottoinsieme è effettivamente istanziabile.	<pre> graph TD     Geometry --&gt; Point     Geometry --&gt; Curve     Geometry --&gt; Surface     Geometry --&gt; GeomCollection     Point --- LineString     Curve --- Polygon     Surface --- Polygons     GeomCollection --- MultiSurface     GeomCollection --- MultiCurve     GeomCollection --- MultiPoint     MultiSurface --- MultiPolygon     MultiSurface --- MultiLineString   </pre>
<b>Feature tables</b>	Tabelle che contengono uno o più attributi geometrici. <b>L'id mi aiuta a gestire</b> ; poi ho gli attributi delle foglie (e poi ha laggato quindi non lo saprò mai)	<pre>CREATE TABLE &lt;feature-name&gt; (   &lt;FIELD name&gt; &lt;FIELD type&gt;,   &lt;feature attributes&gt;,   &lt;other foreign FID&gt; REFERENCES &lt;other feature table&gt;,   &lt;geometry attribute 1&gt; &lt;Geometry type&gt;,   ... (other geometric attributes for feature) )  PRIMARY KEY &lt;FIELD name&gt;, FOREIGN KEY (&lt;foreign FID&gt;, ...) REFERENCES &lt;other feature table&gt; (&lt;other FID name&gt;, ...) )</pre>

#### Rappresentazione testuale

È possibile visualizzare i dati geometrici anche senza interfaccia grafica attraverso la SQL textual representation o geometry. Visualizzo il tipo di geometria e le coordinate.

#### Esempietto

##### Relations

WOOD(Type: STRING, Extension: MULTIPOLYGON)  
MEADOW(Pasture: BOOLEAN, Extension: MULTIPOLYGON)

##### Spatial indices

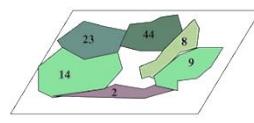
WOOD\_spatial\_idx ON WOOD(Extension);  
MEADOW\_spatial\_idx ON MEADOW(Extension);

Geometry Type	SQL Text Literal Representation	Comment
Point	'POINT (10 10)'	a Point
LineString	'LINESTRING ( 10 10, 20 20, 30 40)'	a LineString with 3 points
Polygon	'POLYGON ((10 10, 10 20, 20 20, 20 15, 10 10))'	a Polygon with 1 exterior ring and 0 interior rings
Multipoint	'MULTIPOINT (10 10, 20 20)'	a MultiPoint with 2 point
MultiLineString	'MULTILINESTRING ((10 10, 20 20), (15 15, 30 15))'	a MultiLineString with 2 linestrings
MultiPolygon	'MULTIPOLYGON (((10 10, 10 20, 20 20, 20 15, 10 10), ((60 60, 70 70, 80 60, 60 60 ))))'	a MultiPolygon with 2 polygons
	(10 10), )'	a GeometryCollection consisting of 2 Point values and a LineString value

Tipo	Extension
Copse	23,44
Broadleaf	8,44

Pasture	Extension
True	14,9
False	2

Graphical representation  
of the geometries



## PostGIS

È l'estensione di postgresSQL che si occupa di **gestire i dati spaziali**. Introduce **tipi geometrici** e **funzioni** per i dati spaziali. È possibile creare una colonna di tipo geometry/specifico

### Generare e importare tabelle

Usiamo ancora il classico **CREATE TABLE** come siamo abituati 😊 Oppure possiamo aggiungere la funzione **AddGeometryColumn**, che aggiunge una colonna geometrica ad una tabella, costringendomi a specificare un sistema di riferimento e aggiornando la metainformazione nella tabella **Geometry Column** affinché tutti i sistemi che si interfacciano possano sapere dove sono i dati.

Posso tradurre files di tipo shapefile in file sql attraverso il comando da linea di comando

```
shp2pgsql <options> <shapefile> <table> <fileSQL>
```

Con l'output posso poi generare una tabella SQL che contiene i dati contenuti dallo shapefile. Infine, posso travasare questi dati in una tabella più tradizionale.

**! Le geometrie potrebbero non essere ben formate per SQL, dato che lo shapefile non ha vincoli così forti !**

Questi dati sono poco visibili, comunque; quindi esistono alcune **funzioni per visualizzare i dati in modo decente**. Oppure sfrutto altri **software GIS** come OpenJump o QGIS.

PostGIS fornisce anche un centinaio di funzioni che implementano trasformazioni fra sistemi di riferimento, trasformazioni varie e le relazioni tipologiche di Clementini. Sulla documentazione ci sono anche i disegnetti!

<b>boolean ST_Disjoint (g1, g2)</b>	La definizione cambia rispetto alla nostra! Va per negazione: se tutte le altre sono false allora sono in disjoint
<b>boolean ST_Touches (g1, g2)</b>	True se gli unici punti in comune sono nei boundaries
<b>boolean ST_Within(g1,g2)</b>	È diverso; ritorna vero anche quando due geometrie sono uguali
<b>boolean ST_Contains(g1, g2)</b>	È l'inverso del within, e condivide il difetto che se sono uguali ritorna true.
<b>boolean ST_Equals(g1, g2)</b>	Vero solo se rappresentano lo stesso insieme di punti, anche se la rappresentazione matematica è diversa. Questo significa che se ho un segmento che rappresenta 0,0-2,2 posso rappresentarlo come unico segmento, o come due segmenti uniti; risulta equal. OrderingEquals verifica anche che abbiano la stessa rappresentazione
<b>boolean ST_Overlaps</b> <b>boolean ST_Crosses</b>	Come in Clementini.
<b>float ST_Distance(g1, g2)</b>	Ritorna la distanza.

### Interrogazioni

```
Municipality(ID, Name, Inhabitants,  
Extension: MULTIPOLYGON)  
Province(ID, Name, Extension: MULTIPOLYGON)  
River(ID, Name, Path: MULTILINESTRING)  
Road(ID, Name, Type, Path: MULTILINESTRING)
```

Trovare il nome di tutti i fiumi che attraversano almeno due province diverse.

```
SELECT R.Name  
FROM River, Province P1, Province P2  
WHERE P1.ID <> P2.ID  
      AND ST_Crosses(R.Path, P1.Extension)  
      AND ST_Crosses(R.Path, P2.Extension)
```

→ Devo fare attenzione al significato delle cose, dato che le relazioni topologiche sono molto diverse 😞  
“Attraversa” per noi significa che deve stare un po’ dentro e un po’ fuori.

*Trovare il nome dei comuni che sono localizzati ad una distanza minore di 20Km dal fiume Adige.*

```
SELECT M.Name  
FROM Municipality M, River R  
WHERE R.Name = 'Adige'  
AND ST_Distance(R.Path, M.Extension) < 20
```

```
SELECT M.Name  
FROM Municipality M, River R  
WHERE R.Name = 'Adige' AND  
(ST_Overlaps(M.Extension, ST_Buffer(R.Path, 20.000))  
OR ST_Contains(M.Extension, ST_Buffer(R.Path, 20.000)))
```

→ Anche qui posso procedere in due modi: o uso direttamente la distanza, o uso un buffer per costruire tutti i punti che distano 20 Km dall'Adige e poi chiedo che venga soddisfatta una certa relazione tra il buffer e la municipality. Attenzione: overlap garantisce che la distanza sia 20 Km, però escludo il caso in cui il comune sia interamente a meno di 20 Km (=sia contenuto nel buffer). Quindi metto il contains in or.

Dal punto di vista dell'efficienza è circa uguale: distanza e overlap vanno entrambi in  $n \log n$ . Tuttavia, se uso la distanza devo calcolare la distanza per tutti i comuni e poi rispondere; invece con overlap potrei sfruttare qualche indice.

L.20 – 23/05/2022

*Trovare per ogni strada i fiumi che interseca, ritornando nel risultato il nome della strada, del fiume e la geometria WKT della strada.*

```
SELECT RD.Name, RV.Name, ST_AsText(RV.Path)  
FROM Road RD, River RV  
WHERE NOT (ST_Disjoint(RD.Path, RV.Path))  
OR (ST_Equals(RD.Path, RV.Path))
```

→ Negare la disjoint è il modo più facile e veloce per indicare un qualunque tipo di intersezione. Resta fuori l'uguaglianza: in teoria un fiume non sarà mai uguale a una strada, ma just to be safe lo facciamo notare.

*Per ogni provincia trova il numero di municipi che contiene, ritornando nel risultato il nome della provincia, il numero di municipi e la media degli abitanti di questi municipi.*

```
SELECT M.Name, AVG(M.Inhabitants) AS AverageInhabitants, COUNT(*) AS NumberOfMunicipalities  
FROM Municipality M, Province P  
WHERE ST_Contains(M.Extension, P.Extension)  
GROUP BY P.ID, P.Name
```

→ Devo mettere anche il Name nel GROUP BY così ce l'ho riportato nella tabella finale.

*Trova le municipalità con meno di 10.000 abitanti che sono adiacenti a una municipalità con più di 1000000 abitanti, ritornando nel risultato il nome della municipalità e il nome della sua provincia.*

**Proposta mia: metto tutte e tre le tabelle nella from e poi metto la condizione sulle tre.  
Dice che va bene ma lui fa nell'altro modo, con l'EXISTS.**

```
SELECT Msmall.Name, P.Name  
FROM Municipality Mbig, Municipality Msmall,  
Province P  
WHERE ST_Touches(Mbig.Extension, Msmall.Extension)  
AND ST_Contains(Msmall.Extension, P.Extension)  
AND Msmall.Inhabitants < 10.000  
AND Mbig.Inhabitants > 1.000.000
```

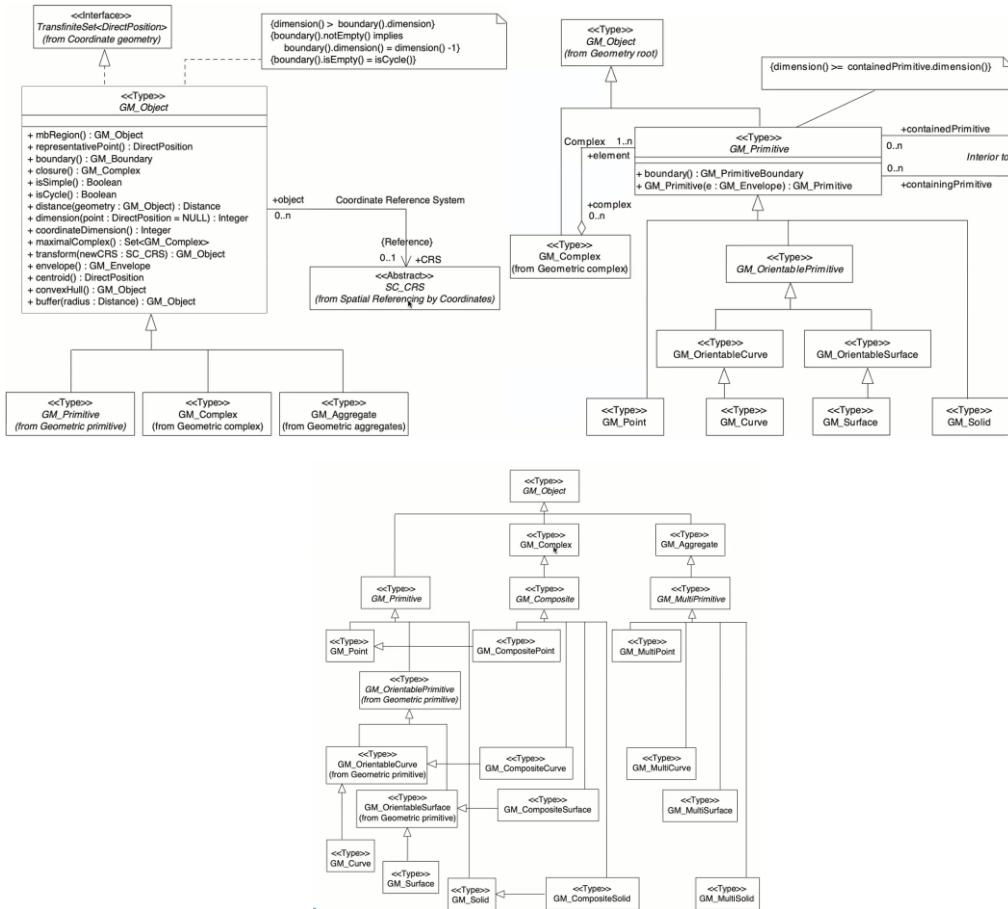
```
SELECT M.Name, P.Name  
FROM Municipality M, Province P  
WHERE M.Inhabitants < 10.000  
AND ST_CONTAINS(P.Extension, M.Extension)  
AND EXISTS (  
SELECT 1  
FROM Municipality M1  
WHERE M1.Inhabitants > 1.000.000  
AND ST_Touches(M.Extension, M1.Extension)  
)
```

*Ha detto che non chiede cose più complicate di questa. :)*

# Modellazione del dato spaziale in UML

Precisazione: per modellare a livello concettuale una DB che contiene info spaziale senza fare riferimento a uno specifico sistema abbiamo bisogno di una rappresentazione astratta dei tipi, da usare in UML.

Possiamo usare lo standard ISO "Spatial Schema", che fornisce un insieme di datatype UML che descrivono con vari approcci l'info spaziale. L'idea è sempre di mettersi nel modello vettoriale



**"GLI STANDARD ISO SONO A PAGAMENTO, VI COSTANO 100 EURO A DOCUMENTO, NON SI SA BENE PERCHÉ.**

POI INSOMMA SI TROVANO ANCHE... PER ALTRE VIE."

Negli esercizi usiamo queste corrispondenze:

“ Questa slide se la dimenticano tutti, i vostri collegh prima di noi.” 😞

Non mi mettete "stringa", mi mettete quelli a sinistra."

Esiste anche lo standard INSPIRE, che vuole definire un formato di scambio comune fra stati europei; usa l'ISO dell'epoca per essere il più possibile compatibile e universale. Ne esiste anche una rappresentazione XML per essere indipendente dai sistemi.

Libreria Java di riferimento

Tutta questa bella roba è disponibile in Java attraverso la libreria JTS Topology suite.

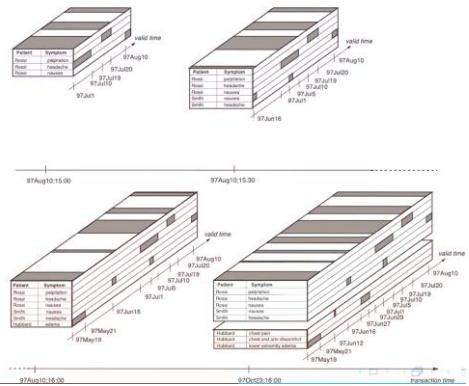
UML (ISO spatial schema)	OGC (simple feature specification)
GM_Object	
GM_Aggregate	➔ GEOMETRYCOLLECTION
GM_Curve	➔ LINESTRING
GM_MultiCurve	➔ MULTILINESTRING
GM_Point	➔ POINT
GM_MultiPoint	➔ MULTIPOINT
GM_Surface	➔ POLYGON
GM_MultiSurface	➔ MULTIPOLYGON

## DATABASE TEMPORALI

I database temporali sono richiesti in maniera abbastanza pesante; la gestione del tempo consente di rispondere a richieste importanti.

Perché c'è bisogno di una soluzione particolare? Il tempo può essere definito già prima che venisse definito un DB specifico; i relazionali hanno già tipi di data o timestamp. Se questo mi è sufficiente, ottimo!

Un modo completamente diverso di gestire il tempo, invece, è quella di ritenere che il **tempo sia una dimensione ortogonale al resto dell'informazione**, che caratterizza qualsiasi istanza che rappresento nella base di dati. Questa considerazione fa nascere le basi di dati temporali, nelle quali rappresentiamo il tempo come caratterizzante qualsiasi attributo e qualsiasi istanza di informazione.



**Def: VT – Valid time**

Tempo associato all'istanza così come questa è nella realtà, nel contesto applicativo che sto descrivendo.

**Def: TT – Transaction time**

Informazione temporale che descrive la mia conoscenza di quell'informazione, ovvero quando io so che quella cosa è in un certo modo.

Li distinguo perché in generale **non riesco sempre a far coincidere il tempo in cui l'informazione nasce con il tempo in cui lo registro nella DB**.

Il **valid time** è problematico, poiché **non è sempre spiegato bene**. In generale, questi sistemi sono usati in ambito medico – e dunque si fa riferimento alla comparsa di un evento per un paziente (aka è abbastanza ovvio). Se vado in altri contesti – ad esempio “la persona interagisce con l'organizzazione” – non è più banale! Tipicamente si mette quando nasce ed è aperto fino a quando muore, oppure legarlo al momento di interazione con un'org (ad esempio immatricolazione o periodo di ricovero). Nzomma, va definito e non è automatico.

Il **transaction time**, invece, è abbastanza chiaro: è il tempo durante il quale è noto alla DB.

Nota: dato che devo sapere quando finisce il transaction time, **in realtà nelle DB temporali le istanze di informazione non vengono mai cancellate**.

Ogni volta che faccio una modifica, quello che faccio è **chiudere l'informazione vecchia e ne apro un'altra con quella aggiornata**. Questo mi permette di ricostruire qual era l'informazione nota di quel paziente in un certo momento del tempo.

In pratica, a livello di tupla (o di attributo) devo aggiungere un campo con il VT e TT.

Patient	Sympthom	VT	TT	Patient	Sympthom	VT	TT
Rossi	Palpitations	[19/07/97 20:30, 20/07/97 07:00]	[10/08/97 15:00, ∞]	Hubbard	Oedema	[19/05/97 17:00, 21/05/97 09:00]	[10/08/97 16:00, 23/10/97 16:00]
Rossi	Headache	[01/07/97 20:45, 10/07/97 13:00]	[10/08/97 15:00, ∞]	Hubbard	Lower extremity oedema	[19/05/97 17:00, 21/05/97 09:00]	[23/10/97 16:00, ∞]
Rossi	Nausea	[10/08/97 10:00, 10/08/97 14:00]	[10/08/97 15:00, ∞]	Hubbard	Chest pain and arm pain	[27/06/97 08:00, 29/06/97 22:00]	[23/10/97 16:00, ∞]
Smith	Nausea	[16/06/97 17:30, 16/06/97 19:45]	[10/08/97 15:30, ∞]	Hubbard	Chest pain	[12/06/97 08:30, 12/06/97 09:45]	[23/10/97 16:00, ∞]
Smith	Nausea	[05/07/97 09:00, 05/07/97 17:30]	[10/08/97 15:30, ∞]				
Hubbard	Oedema	[19/05/97 17:00, 21/05/97 09:00]	[10/08/97 16:00, ∞]				

- Graficamente posso immaginare di avere una terza dimensione: per il valid time e il transaction time.
- In questa rappresentazione **cambia anche il concetto di chiave**. Una volta che ho un supporto di VT, non ho più bisogno di fare modellazione specifica per lo storico (es. mettere il tempo nell'identificatore come avevamo visto!!)

### Classificazione

I ricercatori si sono sbizzarriti a trovare varie definizioni e varianti del caso, in base a certe caratteristiche. Anche in base alla relazione fra i vari VT e TT possiamo avere delle relazioni (comunque meno complesse rispetto allo spazio, ma still): se esistono certe relazioni posso avere certe caratterizzazioni del DB. Per esempio, se il VT inizia sempre prima del TT time, allora la DB è retroattiva – ovvero registra eventi già iniziati. Se deve passare un certo delta prima di registrare si parla di delayed retroactive.

Dunque, inserite le dimensioni del VT e TT, generiamo i seguenti casi:

- **Snapshot**: registro solo il current state.

- **Valid-time databases** / Historical databases: supporto solo il valid time
- **Transaction-time databases**: supportano solo il transaction time. Permettono di sapere cosa conteneva il DB in ogni momento.
- **Bi-temporal databases**: supportano sia TT che VT.

## Databases bitemporali

Cosa troviamo nei sistemi? Tendenzialmente è più semplice ma non ci si è riusciti a mettere sullo standard; in postgres si trova del supporto ma non è standard.

In generale, posso specificare il tempo come:

- **Istante di tempo**
- **Intervallo di tempo**: usiamo intervalli che possono essere aperti a destra (ovvero "l'intervallo è attivo"), e viene indicato con il simbolo di infinito:  $[1/1/2000, \infty]$
- **Insieme di intervalli** (temporal element-based)

Il tempo viene rappresentato con sistemi di riferimento che possono avere unità diverse. Mentre lo spazio usa sempre i metri, fondamentalmente, qui ho il problema della unità di misura 😞

**Non è banale decidere la granularità**, perché poi rischio di non poter ottenere le info che voglio, o avere granularità diverse e quindi bagoli a integrare le informazioni.

### Relazioni di Allen

Anche qui, dunque, abbiamo bisogno di relazioni temporali; abbiamo relazioni di ordinamento facili sugli istanti di tempo, ma per gli intervalli si rendono necessarie. Sono introdotte dalle relazioni di Allen.

A  -----  B  -----	A IS EQUAL TO B B IS EQUAL TO A
A  -----  B  ----- -----	A IS BEFORE B B IS AFTER A
A  -----  B  ----- -----	A MEETS B B IS MET BY A
A  -----  B  ----- -----	A OVERLAPS B B IS OVERLAPPED BY A
A  -----  B  ----- -----	A STARTS B B IS STARTED BY A
A  -----  B  ----- -----	A FINISHES B B IS FINISHED BY A
A  -----  B  ----- -----	A IS DURING B B CONTAINS A

Nessun sistema implementa esattamente queste relazioni; sono solo la relazione teorica.

## Integrazione di spazio e tempo in Postgres

Per semplicità, ipotizziamo di rappresentare solo il VT – ovvero che ci sono due attributi StartVT e EndVT per ogni tupla.

- Se non ho istanza di fine, sarà a null.
- Nello scrivere query possiamo usare la funzione OVERLAPS, che lavora su intervalli. Posso costruire intervalli mettendo due istanti di tempo fra parentesi, oppure come INTERVAL '100 giorni'.
- Gli intervalli sono creati come semiaperti, ovvero start  $\leq$  time  $<$  end (end non è incluso).
- Se start e end coincidono allora viene rappresentato l'istante di tempo. OVERLAPS è vero se l'intersezione dei due intervalli non è vuota.

Altre operazioni:

date + integer	date	date – date	integer
----------------	------	-------------	---------

date + interval	timestamp		date – integer	date
date + time	timestamp		date – interval	timestamp
interval + interval	interval		time – time	interval
timestamp + interval	interval		time – interval	time
timestamp + interval	timestamp		timestamp – interval	timestamp
time + interval	time		interval – interval	interval
			timestamp – timestamp	interval

## Query temporali

Trova nome e cognome dei pazienti che il 3/2/14 erano stati nel dipartimento di Medicina per almeno 3 giorni e hanno avuto una terapia di ‘paracetamolo’ nello stesso giorno.

```
SELECT P.Name, P.Surname
FROM Patient P, Therapy T
WHERE T.Patient = P.ID AND P.Dept = 'Medicine' AND
      P.StartVT <= DATE '3/2/2014' - INTERVAL '3' DAY
      AND P.EndVT > DATE '3/2/2014'
      AND T.Therapy = 'paracetamol'
      AND (T.StartVT, T.EndVT + 1) OVERLAPS (DATE '3/2/14', DATE '3/2/14')
```

L.21 – 24/05/2022

Trova cognome e nome dei pazienti che erano in un qualche dipartimento il 1/5/15 e che durante la permanenza hanno presentato febbre seguita da sintomi di dolore addominale.

```
SELECT P.Name, P.Surname
FROM Patient P, Symptom S1, Symptom S2
WHERE (P.StartVT, P.EndVT) OVERLAPS (DATE '1/5/15', DATE '1/5/15')
      AND S1.Patient = P.ID AND S2.Patient = P.ID
      AND S1.EndVT <= S2.StartVT
      AND S1.Symptom = 'fever' AND S2.Symptom = 'abdominal pains'
      AND (P.StartVT, P.EndVT+1) OVERLAPS (S2.StartVT, S2.StartVT)
      AND (P.StartVT, P.EndVT+1) OVERLAPS (S2.EndVT, S2.EndVT)
      AND (P.StartVT, P.EndVT+1) OVERLAPS (S1.StartVT, S1.StartVT)
      AND (P.StartVT, P.EndVT+1) OVERLAPS (S2.EndVT, S2.EndVT)
```

**! È ambiguo: cosa significa “seguito da”? Intendiamo che il secondo inizia dopo la fine del primo? Sì dai.  
Lo scrive con l’overlaps, ma si può fare anche col maggiore/minore. !**

(NEXT) Trova il cognome e il nome del paziente che ha avuto una terapia di paracetamolo, e il primo sintomo comparto dopo la terapia è abdominal pain.

```
SELECT p.Name, p.Surname
FROM Patient P, Therapy T, Symptom S
WHERE S.Patient = P.PID AND T.Patient = P.PID
      AND T.Therapy = "paracetamol"
      AND S.Symptom = "abdominal pains"
      AND T.StartVT <= S.StartVT AND //terapia inizia prima del sintomo;
                                         //fermandomi qui avrei TUTTI i sintomi che seguono.
      NOT EXISTS (SELECT 1 FROM SYMPTOM S1
                  WHERE S1.Patient = P.ID AND // Non devono esisterne altri in mezzo.
                                         T.StartVT <= S1.StartVT AND S1.StartVT < S.StartVT)
```

Quantificazione universale: trovare nome e cognome dei pazienti che hanno avuto esattamente gli stessi sintomi il 3/2/14  
Procedo trovando le coppie di pazienti che hanno un sintomo in comune il giorno tal dei tali, e poi dico che non esistono altri sintomi diversi sempre in quei giorni. Udiosanto.

```
SELECT P1.Surname, P1.Name, P2.Surname, P2.Name
FROM Patient P1, Patient P2, Symptom S1, Symptom S2
WHERE S1.Patient = P1.PID AND S2.Patient = P2.ID
      AND S1.Symptom = S2.Symptom AND P1.PID <> P2.PID
      AND (S1.StartVT, S1.EndVT+1) OVERLAPS ('3/2/14, 3/2/14')
      AND (S2.StartVT, S2.EndVT+1) OVERLAPS ('3/2/14, 3/2/14')
      AND NOT EXISTS(
```

```

        SELECT 1 FORM Symptom SS1, Symptom SS2
        WHERE SS1.Patient = P1.PID AND SS2.Patient = P2.PID
              AND (SS2.StartVT, SS2.EndVT+1) OVERLAPS ('3/2/14, 3/2/14')
              AND (SS1.StartVT, SS1.EndVT+1) OVERLAPS ('3/2/14, 3/2/14')
              AND SS1.Symptom <> SS2.Symptom
    )

```

## Query spaziotemporali

---

Una tipologia di info che si presta molto bene è proprio l'evoluzione di una epidemia sul territorio. Ogni istanza della tabella epidemica nasce da un punto.

<pre>CREATE TABLE Epidemic (     Code CHAR(3),     FirstCase POINT,     Disease VARCHAR (50),     StartVT TIMESTAMP,     EndVT TIMESTAMP )</pre>	<pre>CREATE TABLE NumberOfCases (     EpiCode CHAR(3),     Number INTEGER,     StartVT TIMESTAMP,     EndVT TIMESTAMP )</pre>	<pre>CREATE TABLE Evolution(     EpidCode CHAR(3)     Extension MULTIPOLYGON,     StartVT TIMESTAMP,     EndVT TIMESTAMP )</pre>
--	---	--

*Trovare codice e malattia della epidemia attualmente attiva che intersezione il territorio descritto dalla geometria con rappresentazione WKT POLYGON(...)*

```

SELECT E.Code, E.Disease
FROM Epidemic E, Evolution EV
WHERE E.Code = EV.EpidCode
      AND (E.StartVT, E.EndVT + 1) OVERLAPS (DATE'today', DATE'today')
      AND (EV.StartVT, EV.EndVT + 1) OVERLAPS (DATE'today', DATE'today')
      AND NOT ST_Disjoint(EV.Extension, ST_GeomFromText(...))

```

*Trovare l'evoluzione dell'estensione dell'epidemia di codice E345 da 1/1/2015 a 30/4/2015 ritornando la geometria come WKT e l'istante di fine e inizio di ciascuna geometria dell'evoluzione.*

```

SELECT ST_AsText(EV.Extension), EV.StartVT as start, EV.EndVT as end
FROM Evolution EV
WHERE EV.EpidCode = 'E345' // altro modo, senza overlap
      AND NOT (EV.EndVT < DATE '1/1/2015' OR DATE '30/4/2015'<EV.StartVT)
ORDERED BY EV.StartVT

```

*Trovare code e starting e ending time dell'epidemia che non ha mai intersevato il territorio POLYGON(...)*

```

SELECT E.Code, E.StartVT as start, E.EndVT as end
FROM Epidemic E
WHERE NOT EXISTS (
    SELECT 1 FROM Evolution EV
    WHERE E.Code = EV.EpidCode AND NOT ST_Disjoint(EV.Extension, ST_GeomFromText(...)))
)
```

## DIPENDENZE FUNZIONALI E FORME NORMALI [X]

### Introduzione

Sono uno strumento *formale* introdotto per dare un contesto *formale* che consentisse di definire in modo *formale* le **dipendenze fra dati**, che stanno alla base di ridondanza e del concetto di chiave (o meglio, di identificare) ma che può essere usato anche al di fuori del contesto del modello relazionale, in quanto **semplice modo per identificare legami**: può essere applicato ovunque io abbia collezione di istanze di info con attributi.

### Normalization theory

Gli obiettivi della teoria di normalizzazione sono:

- **Trovare la ridondanza nei dati ed eliminarla**
- **Descrivere connessioni semantiche fra i dati**; sono strumenti alternativi a UML/ER per la rappresentazione del modello concettuale
- **Valutazione della qualità di uno schema relazionale**, secondo il criterio tradizionale che vuole ridurre al minimo la ridondanza.
- **Proposte di intervento su schemi per ridurre la ridondanza.**

### Ridondanza

Posso avere più collezioni di informazioni che finiscono in una tabella, e che mi portano a dover riscrivere più volte nella tabella. La ridondanza è **sia un problema che un'opportunità**, in base al sistema che sto considerando; l'inconsistenza mi porta ad anomalie di aggiornamento, inserimento e cancellazione.

Le dipendenze funzionali hanno l'obiettivo di **descrivere le caratteristiche dell'informazione proprio in questo senso – "certe caratteristiche ne determinano altre"** – e questo mi permette di usare meccanismi automatici che verificano la presenza di ridondanza e l'applicazione di variazioni allo schema per evitarla o accorgimenti per mantenerla consistente. Inoltre, possiamo precisare anche vincoli di integrità.

### Dipendenza funzionale

Consideriamo  $r$  un'istanza di una relazione con schema  $R(X)$ , e chiamiamo due sottoinsiemi  $\alpha \subseteq X$  e  $\beta \subseteq X$ .

Diciamo che  $\alpha$  **dermina funzionalmente**  $\beta$  ( $\alpha \rightarrow \beta$ ) in  $r$  se, per ogni coppia  $t_1, t_2 \in r$  che hanno lo stesso valore in  $\alpha$ , rappresentano lo stesso valore anche in  $\beta$ .

Più formalmente:

$$\alpha \rightarrow \beta \text{ è soddisfatto in } r \text{ se } \forall t_1, t_2 \in r ((t_1[\alpha] = t_2[\alpha]) \Rightarrow (t_1[\beta] = t_2[\beta]))$$

! Questa proprietà è una proprietà generale del dato; se è vero che il codice fiscale determina cognome e nome, è vero su tutte le istanze. Derivano dai requisiti. !

Una dipendenza fuzionale  $\alpha \rightarrow \beta$  è **valida sullo schema**  $R(X)$  se per ogni istanza  $r$  di  $R(X)$  è vero che  $r$  soddisfa  $\alpha \rightarrow \beta$ .

#### Esempio

*EXAM (Code, EType, Surname, Name, BDate, Result, EDate)*

Attenzione: queste frecce significano solamente che queste cose dipendono dalle altre, non necessariamente che troverò questa ridondanza.

#### Example 2

D<sub>1</sub>: Code → Surname Name BDate  
D<sub>2</sub>: Code EType EDate → Result  
D<sub>3</sub>: Surname Name → Name

### Derivazione (parentesi)

In generale, posiamo dire che **partendo da un nucleo di dipendenze funzionali che esprimiamo su un certo insieme di dati se ne possono derivare altre per inferenza**. Quelle derivate sono logicamente implicate dalle dipendenze funzionali iniziali.

Dato un certo insieme di dipendenze funzionali  $\mathcal{F}$  posso dire che da  $\mathcal{F}$  ottengo altre dipendenze funzionali, che sono logicamente implicate da  $\mathcal{F}$ .

$$\mathcal{F} \models \alpha \rightarrow \beta$$

Esistono un insieme di **assiomi** che mi permettono di fare queste derivazioni e che derivano dalla definizione di dipendenza funzionale. Non li vediamo.

*Def: Chiusura di una dipendenza funzionale*

L'insieme di tutte le dipendenze funzionali che possono essere derivate si chiama **chiusura di una dipendenza funzionale**; si indica con  $\mathcal{F}^+$ , e indica tutto l'insieme di conoscenza che abbiamo partendo dalla dipendenze funzionali assegnate.

*Esempietto*

Partendo con:

$$\begin{aligned}D_1: \text{Code} &\rightarrow \text{Surname Name BDate} \\D_2: \text{Code EType EDate} &\rightarrow \text{Result} \\D_3: \text{Surname Name} &\rightarrow \text{Name}\end{aligned}$$

Ho che:

$$\begin{aligned}D_1, D_2 \models \text{Code Etype EDate} &\rightarrow \text{Surname Name BDate Result} \\D_1, D_2 \models \text{Code EType EDate} &\rightarrow C\end{aligned}$$

Quindi,  $(\text{Code}, \text{Etype}, \text{EDate})$  sono una **superchiave** della tabella *EXAM*, dato che determinano tutti gli attributi della tabella (aka due righe che hanno gli stessi valori in code, type, date hanno anche gli stessi valori di tupla).

**Chiavi e superchiavi**

Le dipendenze funzionali sono anche un altro modo di **definire il concetto di chiave e superchiave**.

*Def: Superchiave*

Dato uno schema relazionale  $R(X), K \subseteq X$  è detto **superchiave** di  $R(X)$  se  $K$  determina funzionalmente  $X (K \rightarrow X)$  – ovvero **o è già nelle dipendenze oppure deve essere derivabile**.

*Def. Chiave*

Ho una **chiave** quando  $K$  è **minimale**, ovvero data una relazione con schema  $R(X), K \subseteq X$  è una chiave di  $R(X)$  se  $K$  determina funzionalmente  $X (K \rightarrow X)$  e non esiste un sottoinsieme proprio  $\alpha \subset K$  tale per cui  $\alpha$  determina  $X (\alpha \rightarrow X)$ .

## Forme normali: BCNF

Il primo processo da definire è quello di definire la conoscenza, ovvero **anziché fare lo schema concettuale specifico come gli attributi sono fra loro dipendenti**. L'obiettivo era di valutare gli schemi relazionali e verificare la quantità e tipologia di ridondanza presente. **Le forme normali definiscono uno stato** (es. assenza di ridondanza, ridondanza di un certo tipo...) e permettono di introdurre un meccanismo per decomporre uno schema per ottenere una forma normale se questa non è soddisfatta.

In generale, decomporre significa **spezzare uno schema**, tirar fuori una lista di dipendenze che stava dentro, etc.

**Definizione**

Una relazione  $R(X)$  è in forma normale Boyce e Codd se, rispetto a un insieme di dipendenze funzionali  $\mathcal{F}$ , per ciascuna dipendenza funzionale non triviale  $\alpha \rightarrow \beta \in \mathcal{F}^+$  che è valida in  $X$  è vero che:

$$\alpha \rightarrow X \in \mathcal{F}^+, \text{ oppure } \alpha^+ = X$$

**Ovvero  $\alpha$  è superchiave per  $R(X)$ .**

Questa definizione vale su una relazione; se voglio dirlo su un intero schema basta che ogni sua relazione sia in BCNF. È la più restrittiva. Calcolare  $\mathcal{F}^+$  è complesso, dato che di mezzo c'è anche il punto fisso. Un escamotage è quello della chiusura degli attributi alpha.

*Def: Chiusura di un insieme di attributi*

Dato un insieme di dipendenze funzionali  $\mathcal{F}$  su un insieme di attributi e un  $\alpha \subseteq X$ , chiamiamo chiusura di  $\alpha$  rispetto a  $\mathcal{F}$  il seguente insieme di attributi

$$\alpha^+ = \{ A \mid \alpha \rightarrow A \in \mathcal{F}^+ \text{ e } A \text{ è un attributo singolo} \}$$

Si parte dall'insieme  $\alpha$ , che sicuramente appartiene a  $\alpha^+$ , e si attivano tutte le dipendenze funzionali a disposizione e che hanno a sinistra attributi di  $\alpha$ . Fatte queste, riparto e vado avanti di nuovo fino a non aggiungere nulla 😊

**Algoritmo per verificare se è BCNF**

È molto lungo :) Parterebbe da  $\mathcal{F}$  ma lavoro sulla chiusura degli attributi per considerare implicitamente  $\mathcal{F}^+$

- Per ogni relazione:
  - Per ogni sottoinsieme  $Y$  di  $X_i$  calcolo  $Y^+$ , che indica cosa determina  $X$  rispetto a tutte le dipendenze funzionali che ho.
    - Se questo  $Y^+$  include  $X_i$ , allora  $Y$  è superchiave e questo va bene.
    - Va bene anche se ottengo solo attributi che non stanno in  $R$  (= è nella tabella ma non determina niente; determina eventualmente attributi di altre tabelle)
    - Else, non è BCNF

Poi ecco: non parto in quarta a considerare *tutti* i sottoinsiemi come scritto qui; è ovvio che verificherò i casi che so essere significativi.

#### Esempio

Dati gli attributi  $X = \{A, B, C, D, E\}$

E le dipendenze funzionali  $\mathcal{F} = \{A \rightarrow B, B \rightarrow C, B \rightarrow D\}$

Verificare se lo schema  $R_1(A, C, E), R_2(B, D)$  è BCNF

#### Algorithm 1 (BCNF Test)

```

In:  $S = \{R_1(X_1), \dots, R_n(X_n)\}$ , a set of functional dependencies  $\mathcal{F}$ 
Out: TRUE/FALSE
Begin
  ForEach  $R_i(X_i) \in S$  do
    ForEach  $Y \subseteq X_i$  do
      compute  $Y^+$ 
      If  $X_i \subseteq Y^+$  or  $(Y^+ \cap (X_i - Y) = \emptyset)$ 
        continue;
      Else
        Return FALSE;
      End if
    End do
  End do
Return TRUE;

```

#### Relazione 1:

Ho una dipendenza funzionale su A, quindi "si attiva" A. Calcolo  $A^+$  aggiungendo, per ogni dipendenza funzionale che ha A a sinistra, le cose a destra:

- $\{A\} \rightarrow attivo A \rightarrow \{A, B\} \rightarrow attivo A, attivo B \rightarrow \{A, B, C, D\}$ 
  - $X_i \subseteq Y^+$ ? In questo caso  $A, C, E \subseteq \{A, B, C, D\}$ ? NO, perché manca la E! Non vale la prima condizione.
  - $Y^+ \cap (X_i - Y) = \emptyset$ ? In questo caso,  $\{A, B, C, D\} \cap ((A, C, E) - (Y)) \neq \emptyset$   
Y determina un pezzo di R1, non tutto, quindi non va bene.
- $\{C\}$  nom c'è
- $\{E\}$  non c'è

= R1 non è BCNF.

#### Relazione 2 :

Per ogni attributo:

- $B^+ = \{B\} \rightarrow \{B, C, D\}$ 
  - $X_i \subseteq Y^+$ ? Sì
- $D^+ = \{D\} \rightarrow \{D\}$ 
  - $X_i \subseteq Y^+$ ?  $(B, D) \subseteq (D)$ ? NOPE
  - $Y^+ \cap (X_i - Y) = \emptyset$ ?  $\{D\} \cap \{\} = \emptyset$  Yep

= R2 è BCNF

Non essendo BCNF significa che ho ridondanza; in effetti, qui ho ridondanza su C, perché *tbh non ho capito un cazzo*.

Da qui in avanti ho deciso che non avevo più voglia, scusate. I did my best.

Def: Decomposizione

Dato uno schema relazionale  $R(X)$ , definiamo decomposizione di  $R(X)$  un insieme di relazioni

$$\{R_1(X_1), R_2(X_2) \dots R_n(X_n)\}$$

Tali per cui  $X = X_1 \cup X_2 \cup \dots \cup X_n$ .

Notare che non è richiesto che le  $X_i$  siano disgiunte fra loro.

Non tutte le decomposizioni sono accettabili; devono avere due proprietà.

#### 1. Essere lossless

##### Theorem 1 (Lossless join decomposition of two relations)

A decomposition  $R_1(X_1), R_2(X_2)$  of a relation  $R(X)$  is a lossless join decomposition w.r.t. a set of functional dependencies  $\mathcal{F}$ , iff one of the following conditions holds:

- $X_0 \rightarrow X_1 \in \mathcal{F}^+$
- $X_0 \rightarrow X_2 \in \mathcal{F}^+$

with  $X_0 = X_1 \cap X_2$

2. **Conservare le dipendenze funzionali;** la divisione mi consente comunque di verificare le dipendenze considerando una sola tabella alla volta.

L'algoritmo di decomposizione BCNF a volte non rispetta questa cosa.

#### Algoritmo per scomposizione

A fronte di una relazione non BCNF vado a vedere le dipendenze funzionali che creano il problema, e tolgo quelle e le aggiungo a una relazione che contiene la decomposizione. (????)