# Basi 2

**Legenda**

- Lettere maiuscole all'inizio dell'alfabeto, es. $A, B, C$ attributi
- Lettere maiuscole alla fine dell'alfabeto, es. $T, X, Y$ : insiemi di attributi
- $R(T)$: schema di relazione costruito sull'insieme di attributi $T$
- $r$: istanza dello schema di relazione $R(T)$, cioè un insieme di ennuple
- $t, u, v$: ennuple di un'istanza di un dato schema di relazione
- $t\{X\}$: ennupla ottenuta da $t$ considerando i soli attributi in $X$

# Dipendenze funzionali

> Data una tabella $R(T)$, e siano $X$, $Y$ due insiemi di attributi non vuoti tali che $X \cup Y \subseteq T$, una dipendenza funzionale è un qualsiasi vincolo della forma $X \to Y$.

In altre parole: una dipendenza funzionale stabilisce una relazione tra attributi di una tabella. Dati gli insiemi di attributi $X$ e $Y$ della tabella $R(T)$ (dove $T$ è l'insieme di tutti gli attributi della tabella, mentre $X$ e $Y$ sono sottoinsiemi non vuoti di $T$), se scrivo $X \to Y$ significa che l'insieme $X$ *implica* l'insieme $Y$, ovvero conoscendo $X$ posso determinare in modo univoco il valore di $Y$.

Un esempio: In una tabella "Studenti", con gli attributi "Matricola", "Nome" e "CorsoDiLaurea", la dipendenza funzionale potrebbe essere Matricola \rightarrow Nome. Ciò significa che dato il valore della matricola di uno studente, possiamo determinare in modo univoco il suo nome. Tuttavia, non possiamo determinare la matricola di uno studente conoscendo solo il suo nome.

## Soddisfacibilità

> Un'istanza **r** di $R(T)$ soddisfa la dipendenza funzionale $X \to Y$ se e solo se ogni coppia di ennuple in r che *coincide* su $X$ coincide anche su $Y$. Formalmente chiediamo
> $\forall t_1, t_2 \in r : t_1\{X\} = t_2\{X\} \Rightarrow t_1\{Y\} = t_2\{Y\}$.

In altre parole: la soddisfacibilità di una dipendenza funzionale si riferisce alla sua validità o attuabilità all'interno di un insieme di dati, ovvero se la dipendenza funzionale è soddisfatta per tutte le istanze (righe) della tabella $R(T)$.

Scriveremo $R(T, F)$ per indicare uno schema di relazione $R(T)$ con le dipendenze funzionali $F$ che tutte le sue istanze valide devono soddisfare.

| Nome | Telefono | CodiceLibro | Titolo | Data |
|---|---|---|---|---|
| Rossi Carlo | 75444 | XY188A | Decameron | 07-07 |
| Pastine Maurizio | 66133 | XY090C | Canzoniere | 01-08 |
| Paolicchi Laura | 59729 | XY101A | Vita Nova | 05-08 |
| Paolicchi Luca | 59729 | XY701B | Adelchi | 14-01 |
| Paolicchi Luca | 59729 | XY008C | Amlet | 17-08 |

Esempi di dipendenze funzionali soddisfatte o meno dall'istanza:

✓ NomeUtente \rightarrow Telefono

✓ CodiceLibro \rightarrow Titolo, NomeUtente, Data

✗ Telefono \rightarrow NomeUtente

# Dipendenze Derivate

## Implicazione

> Sia $R(T, F)$ uno schema di relazione. Diciamo che F *implica* logicamente la dipendenza funzionale $X \to Y$, indicato con $F \vdash X \to Y$, se e solo se **ogni istanza valida di** $R(T, F)$ **soddisfa anche** $X \to Y$.

L'implicazione descrive la relazione tra due o più dipendenze funzionali all'interno di F (un qualsiasi schema di dipendenze). Indica che una dipendenza funzionale può essere dedotta o inferita da altre dipendenze funzionali all'interno dello schema.

Essa è espressa come una relazione logica tra due o più dipendenze funzionali, in particolare diciamo che una dipendenza funzionale $A \to B$ *implica* una dipendenza funzionale $C \to D$ *se ogni volta che* $A \to B$ *è soddisfatta in uno schema di dati, allora anche* $C \to D$ *deve essere soddisfatta*.

Ad esempio, consideriamo le seguenti dipendenze funzionali ($F$):

1. $A \to B$
2. $B \to C$

Possiamo dedurre o inferire una terza dipendenza funzionale, $A \to C$, basandoci sulle due dipendenze funzionali sopra. Questo significa che l'implicazione $A \to C$ è valida per questo $F$.

## Derivazione

> Una **derivazione** di $X \to Y$ da F è una sequenza finita di dipendenze funzionali $f_1, \dots f_n$ tale che $f_n = X \to Y$ ed ogni $f_i$ è un elemento di $F$ oppure può essere ottenuta da $f_1, \dots f_{i-1}$ usando una regola di inferenza.

La derivazione è il processo di deduzione logica di una dipendenza funzionale da un insimee di altre dipendenze funzionali. Essa viene utilizzata per inferire o dimostrare l'esistenza di una dipendenza funzionale a partire da un insimee di dipendenze funzionali preesistenti.

Attraverso il processo di derivazione, possiamo dimostrare *l'implicazione* tra due insiemi di dipendenze funzionali. Se un insieme di dipendenze funzionali A implica un'altra dipendenza funzionale B, allora possiamo derivare o dimostrare la dipendenza funzionale B a partire dall'insieme di dipendenze funzionali A.

Il processo di derivazione viene eseguito utilizzando le regole derivate.

# Regole Derivate

## Unione

> Se $X \to Y$ e $X \to Z$ ,allora $X \to YZ$.

Permette di combinare due dipendenze funzionali per ottenere una nuova dipendenza funzionale.

Supponiamo di avere due dipendenze $X \to Y$ e $T \to Z$. Se gli insiemi di attributi $X$ e $T$ sono uguali o si sovrappongono, allora possiamo usare le due dipendenze funzionali per ottenere una nuova dipendenza funzionale che copre entrambi gli insiemi di attributi, quindi se $X \to Y$ e $T \to Z$, dove $X \cap T \neq \emptyset$, allora possiamo unire le due dipendenze per ottenere $X \cup T \to Y \cup Z$.

## Decomposizione

> Se $X \to YZ$ , allora $X \to Y$ .

Questa regola permette di dividere una dipendenza funzionale in due dipendenze funzionali più piccole.

Per esempio, data la dipendenza $A \to B$, e dati due insiemi di attributi $C$ e $D$ tali che:

- $C$ è un sottoinsieme di $A$ ($C \subseteq A$)
- $D$ è un sottoinsieme di $B$ ($D \subseteq B$)

possiamo applicare la regola di decomposizione per ottenere le due dipendenze funzionali risultanti:

- $C \to D$
- $(A - C) \to (B - D)$

## Indebolimento (estensione)

> Se $X \to Y$ , allora $XZ \to Y$.

Stabilisce che se una dipendenza funzionale $X \to Y$ è vera, allora possiamo estendere l'insieme di attributi $X$ aggiungendone altri ($Z$) senza modificare la dipendenza funzionale. In altre parole, possiamo "estendere" l'insieme di attributi $X$ senza influenzare la dipendenza funzionale tra $X$ e $Y$.

# Chiusura

> Dato un insieme $F$ di dipendenze funzionali, la chiusura di $F$ è definita come l'insieme $F^+ = \{X \to Y | F \vdash \ X \to Y\}$.

La chiusura di un insieme di dipendenze funzionali è l'insieme di tutte le dipendenze funzionali che possono essere dedotte o inferite da quel particolare insieme.

## Problema dell'implicazione

> Il problema dell'implicazione corrisponde a decidere, dati $F$ e $X \to Y$ , se $X \to Y \in F^+$ oppure no.

Si noti che calcolare $F^+$ applicando gli assiomi di Armstrong ha costo **esponenziale** nel numero di attributi. Calcolare $F^+$ è quindi un modo **algoritmicamente inefficiente** per risolvere il problema dell'implicazione.

### Chiusura di $X$ rispetto ad $F$

> Sia $R(T, F)$ uno schema di relazione. Dato $X \subseteq T$, la **chiusura di $X$ rispetto ad $F$** è definita come $X_F^+ = \{A \in T | F \vdash X \to A\}$.

Essa rappresenta l'insieme di tutti gli **attributi** che possono essere dedotti o inferiti a partire da $X$ utilizzando le dipendenze funzionali in $F$.

Essa viene calcolata iterativamente aggiungendo attributi in base alle dipendenze funzionali in $F$ fino a quando non è più possibile aggiungere nuovi attributi.

Ad esempio, consideriamo uno schema di relazione $R(\{A, B, C, D\})$ con le seguenti dipendenze funzionali:

1. $A \to B$
2. $BC \to D$
3. $D \to A$

Per calcolare la chiusura di $X = \{A\}$, dobbiamo determinare tutti gli attributi che possono essere determinati funzionalmente da A utilizzando le dipendenze funzionali date.

1. Iniziamo con $X = \{A\}$;
2. Applichiamo $A \to B$ e otteniamo $X = \{A, B\}$;
3. Applichiamo $BC \to D$ (siccome $B$ è presente in $X$) e otteniamo $X = \{A, B, D\}$

Non possiamo aggiungere altrii attributi perché non ci sono dipendenze funzionali che coinvolgano $B$, quindi il risultato sarà $X_F^+ = \{A, B, D\}$.

**Soluzione al problema dell'implicazione**

> $F \vdash X \to Y \leftrightarrow Y \subseteq X_F^+$

La dipendenza funzionale $X \to Y$ deriva da $F$ se e solo se $Y$ è un sottoinsieme degli attributi che possono essere determinati funzionalmente da $X$ utilizzando $F$, quindi $Y$ è una chiusura rispetto ad $X_F^+$.

Questo significa che per decidere se $X \to Y \in F^+$ (problema dell'implicazione) si può controllare se $Y \subseteq X_F^+$, che è calcolabile in tempo **polinomiale**.

# Chiavi

## Superchiave

> Dato uno schema di relazione $R(T, F)$, un insieme di attributi $X \subseteq T$ è una **superchiave** di $R$ se e solo se $X \to T \in F^+$.

Insieme di uno o più attributi di una relazione che può essere utilizzato per identificare univocamente ogni tupla all'interno di quella relazione. In altre parole un insieme di attributi $X$ è considerato una superchiave di $R$ se per ogni tupla, i valori degli attributi in $X$ sono diversi: non possono esistere due tuple che abbiano gli stessi valori per gli attributi della superchiave.

# Chiave

> Una chiave è una superchiave minimale, cioè una superchiave tale che nessuno dei suoi sottoinsiemi propri sia a sua volta una superchiave.

È l'insieme di attributi della superchiave più piccolo per garantire l'unicità della tupla. La differenza con la superchiave è che la superchiave può avere elementi ridondanti, per esempio la superchiave composta da "Matricola" e "Numero di telefono" identifica univocamente uno studente, ma lo farebbe anche considerando solo "Matricola", non c'è bisogno del numero di telefono. In questo caso "Matricola" è una chiave.

#### Attributo primo

> Un attributo è primo se e solo se appartiene ad almeno una chiave.

# Verifica di superchiave

Dato $R(T, F)$, possiamo verificare se $X \subseteq T$ è una superchiave tramite il seguente algoritmo di costo **polinomiale**:

1. Calcola la chiusura $X_F^+$
2. Verifica se $X_F^+ = T$

> Sia $G = \{AB \to C, E \to A, A \to E, B \to F\}$. $ABD$ è superchiave?
>
> 1. $ABD_0^+ = ABD$
> 2. $ABD_1^+ = ABDC$ (applicando $AB \to C$)
> 3. $ABD_2^+ = ABDCE$ (applicando $A \to E$)
> 4. $ABD_2^+ = ABDCEF$ (applicando $B \to F$)
>
> $ABD$ è superchiave

# Verifica di chiave

Dato $R(T, F)$, possiamo verificare se $X \subseteq T$ è una chiave tramite il seguente algoritmo di costo **polinomiale**:

1. Verifica se X è una superchiave. Se non lo è, non è una chiave
2. Verifica che per ogni $A \in X$ si abbia $(X - \{A\})_F^+ \neq T$ (quindi rimuovendo uno alla volta gli elementi da $X$, non posso ottenere $T$ facendo la chiusura)

> Sia $G = \{AB \to C, E \to A, A \to E, B \to F\}$.
> $ABD$ è superchiave come dimostrato, ma è anche chiave?
>
> 1. $A$ non può essere rimosso: $BD_G^+ = BDF$
> 2. $B$ non può essere rimosso perché $AD_G^+ = ADE$
> 3. $D$ non può essere rimosso perché $AB_G^+ = ABCEF$

## Trovare una chiave

Dato $R(T, F)$ è possibile trovare una sua chiave in tempo polinomiale. L'idea dell'algoritmo è di partire da T e rimuovere uno ad uno tutti gli attributi non indispensabili per derivare T.

```
function FindKey(R(T,F)):
    K = T
    for all A in T:
        if closure(K - {A}) == T then
            K = K - {A}
    return K
```

> Sia $G = \{AB \to C, E \to A, A \to E, B \to F\}$. Costruiamo una chiave.
>
> 1. $K_0 = ABCDEF$
> 2. $BCDEF_G^+ = ABCDEF$, quindi $BCDEF$ è una superchiave $\to K_1 = BCDEF$
> 3. $CDEF_G^+$ = $ACDEF$, quindi non possiamo rimuovere $B$
> 4. $BDEF_G^+$ = $ABCDEF$, quindi $BDEF$ è una superchiave $\to K_2 = BDEF$
> 5. $BEF_G^+$ = $ABCEF$, quindi non possiamo rimuovere $D$
> 6. $BDF_G^+$ = $BDF$ quindi $E$ va tenuto
> 7. $BDE_G^+$ = $ABCDEF$ quindi $BDE$ è una superchiave e anche una chiave

## Trovare tutte le chiavi

Tuttavia, per calcolare **tutte** le chiavi di uno schema tramite questo algoritmo arriviamo ad un costo **esponenziale**, esiste dunque un algoritmo piuttosto ottimizzato per la ricerca di tutte le chiavi:

```
function FindAllKeys(R(T,F)):
    Z = {B \in T | ∀X \rightarrow Y \in F : B / \in  Y }
    Cand = [Z :: (T-Z)]
    Keys = []
    while Cand != [] do:
        X :: (Y) = Head(Cand)
        Cand = RemoveFirst(Cand)
        if /∃K \in Keys : K ⊂ X then:
            if closure(X, F) = T then:
                Keys = Keys + X
            else:
                A[1],...,A[n] = Y - closure(X, F)
                for i in 1,...,n do:
                    Cand = Cand + (XA[i] :: (A[i+1],...,A[n]))
    return Keys
```

1. L'insieme $Z$ viene inizializzato con tutti gli attributi di $T$ che non compaiono mai come attributi di destinazione $Y$ nelle dipendenze funzionali $F$
2. L'insieme $Cand$ viene inizializzato con una lista contenente un elemento, ovvero $Z$ come chiave candidata iniziale
3. Avvia un ciclo while che continua finché $Cand$ non è vuoto

4. Estrai la prima chiave candidata $X :: (Y)$ dall'insieme Cand
5. Se non esiste alcuna chiave $K$ nelle $Keys$ tale che $K$ è un sottoinsieme proprio di $X$, allora procedi con la verifica della chiave candidata
6. Se $X_F^+ = T$ allora $X$ è una chiave quindi la aggiungi a $Keys$
7. Se $X_F^T \neq T$, trova gli attributi $A_1, \ldots, A_n$ che sono presenti in $Y$ ma non in $X_F^+$, questi attributi saranno gli attributi rimanenti per generare nuove chiavi candidate
8. Per ogni attributo $A_i$, crea una nuova chiave candidata $XA_i :: (A_{i+1}, \ldots, A_n)$ e aggiungila all'insieme $Cand$

# Forma Canonica

Abbiamo visto vari algoritmi che operano sull'insieme delle dipendenze funzionali. Per questo motivo è utile portare tale insieme in una forma più "disciplinata", detta **forma canonica**, equivalente all'originale.

> **Equivalenza**
>
> Due insiemi di dipendenze funzionali $F$ e $G$ sono equivalenti, indicato con $F \equiv G$, se e solo se $F^+ \equiv G^+$.

> **Attributo estraneo**
>
> Sia $X \to Y \in F$, l'attributo $A \in X$ è **estraneo** se e solo se $X\{A\} \to Y \in F^+$.

Attributo di una relazione che non fa parte della chiave primaria. Può essere presente nella relazione ma non contribuisce all'univocità della relazione.

> **Dipendenza ridondante**
>
> La dipendenza $X \to Y \in F$ è ridondante se e solo se $X \to Y \in (F - \{X \to Y\})^+$

Si verifica quando la relazione può essere dedotta o inferita da altre relazioni, contiene dunque informazioni superflue che possono portare a problemi di coerenza dei dati.

> **Forma canonica**
>
> F è in forma canonica se è solo se per ogni $X \to Y \in F$:
>
> 1. $|Y| = 1$
> 2. $X$ non cintiene attributi estranei
> 3. $X \to Y$ non è ridondante

> **Copertura canonica**
>
> $G$ è una copertura canonica di $F$ se e solo se $F \equiv G$ e $G$ è in forma canonica

> **Teorema**
>
> *Per ogni insieme di dipendenze $F$ esiste una copertura canonica.*

```
function Canonize(F):
    G = {X -> A | ∃Y : X -> Y \in F ∧ A \in Y }
    for all X -> A in G such that |X| > 1:
        Z <- X
        for all B in X:
            if A \in closure(Z\{B}, G):
                Z <- Z \ {B}
        G = (G \ {X -> A}) ∪ {Z -> A}
    for all X -> A in G:
        if A \in closure(X, G \ {X->A}):
            G = G \ {X->A}
    return G
```

1. Inizializziamo un insieme di dipendenze $G$ come vuoto
2. Iteriamo su ciascuna dipendenza funzionale $X \to A \in F$ tale che $X$ abbia una cardinalità maggiore di 1 (ossia più di un attributo):
    1. Creiamo un insieme $Z$ che inizialmente è uguale $X$
    2. Iteriamo su ogni attributo $B$ in $X$:
        1. Se $A$ appartiene alla chiusura di $Z - \{B\}$ rispetto all'insieme di dipendenze funzionali $G$, rimuoviamo l'attributo $B$ da $Z$
    3. Aggiorniamo l'insieme di dipendenze funzionali da $G$ rimuovendo la dipendenza funzionale $X \to A$ e aggiungendo la nuova dipendenza funzionale $Z \to A$
3. Iteriamo su ogni $X \to A$ in $G$
4. Verifichiamo se $A$ appartiene alla chiusura di $X$ rispetto all'insieme $G$. Se $A$ appartiene rimuoviamo la dipendenza funzionalae $X \to A$ da $G$
5. Restituiamo l'insieme di dipendenze funzionali $G$, che rappresenta la forma canonica dell'insieme di dipendenze funzionali $F$.

# Decomposizione di Schemi

## Anomalie

Schemi di scarsa qualità soffrono di anomalie, che vanno ad ostacolare le operazioni di inserimento, cancellazione ed aggiornamento dei dati. L'eliminazione di anomalie è tipicamente basata sulla decomposizione di schemi mal definiti in schemi più piccoli, equivalenti ma più disciplinati.

## Proiezione

> Dati uno schema $R(T, F)$ e $Z \subseteq T$, la proiezione di $F$ su $Z$ è definita come l'insieme $\pi_Z(F) = \{X \to Y \in F^+ | X \cup Y \subseteq Z\}$.

La proiezione di F su Z restituitsce un insieme di dipendenze funzionali che coinvolgono solo gli attributi presenti in Z. Questa operazione può essere utile quando si desidera concentrarsi solo sulle dipendenze funzionali che coinvolgono un sottoinsieme specifico di attributi di uno schema, semplificando così l'analisi delle relazioni tra quegli attributi.

È l'insieme delle dipendenze funzionali che soddisfano i seguenti requisiti:

- $X \to Y$ deve appartenere a $F^+$
- L'insieme formato dall'unione di $X$ e $Y$ ($X \cup Y$) deve essere incluso in $Z$

Per esempio, considerando $R(\{A, B, C, D\}, \{AB \to C, CD \to B, B \to D\})$, vogliamo eseguire la proiezione di $F$ sull'insieme $Z = \{A, B, D\}$. Applicando la definizione della proiezione, verifichiamo le dipendenze funzionali di F che soddisfano le condizioni:

- $AB \to C$:
  - Appartiene a $F^+$? Si
  - $AB \cup C \in Z$? No, perché $ABC \notin ABD$

  quindi non verrà inclusa nella proiezione
- $CD \to B$
  - Appartiene a $F^+$? Si
  - $CD \to B \in Z$? No, perché $CDB \notin ABD$

  quindi non verrà inclusa nella proiezione
- $B \to D$
  - Appartiene a $F^+$? Si
  - $B \to D$? Si

  quindi verrà aggiunta alla proiezione

Dunque $\pi_Z(F) = \{B \to D\}$.

# Decomposizione

> Dato uno schema $R(T, F)$, una sua decomposizione è un insieme di schemi $\rho = \{R_1(T_1, F_1), \ldots, R_n(T_n, F_n)\}$ tale che $\cup_i T_i = T, \forall i : Ti \neq \emptyset$ e$\forall i : F_i = \pi_{T_i}(F)$.

Processo di suddivisione di uno schema in più schemi più piccoli. L'obiettivo è quello di migliorare la struttura dello schema, riducendo la ridondanza dei dati, garantendo la non perdita di informazioni e facilitando le operazione di interrogazione e gestione del database.

Durante la decomposizione, si identificano le dipendenze funzionali tra gli attributi dello schema originale e si utilizzano queste dipendenze per suddividere lo schema in relazioni più piccole. Idealmente, le nuove relazioni risultanti dalla decomposizione devono mantenere le stesse informazioni dello schema originale e devono garantire la consistenza dei dati.

La decomposizione di $R(T, F)$ viene indicata come $\rho = \{R_1(T_1), \ldots, R_n(T_n)\}$.

## Preservazione di dati

> **Decomposizione che preserva i dati**
>
> La decomposizione $\rho = \{R_1(T_1), \ldots, R_n(T_n)\}$ preserva i dati se e solo se per ogni istanza $r$ di $R(T, F)$, si ha r = $\pi_{T_1}(r) \rhd\lhd \ldots \rhd\lhd \pi_{T_n}(r)$

Significa che una decomposizione di uno schema preserva i dati se, per ogni istanza presente nello schema originale, la stessa istanza può essere ottenuta combinando le istanze delle relazioni decomposte.

Per verificare se una decomposizione $\rho$ preserva i dati basta che una delle due condizioni sia verificata:

> Sia $\rho = R_1(T_1), R_2(T_2)$ una decomposizione di $R(T, F)$, si ha che $\rho$ preserva i dati se e solo se $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$.

- La dipendenza funzionale $T_1 \cap T_2 \to T_1$ appartiene alla chiusura ($F^+$) delle dipendenze funzionali dell'insieme $F$
- La dipendenza funzionale $T_1 \cap T_2 \to T_2$ appartiene a $F^+$

Questo permette di ricondurre il problema di determinare se una certa decomposizione binaria preserva i dati al problema dell'implicazione, che ha costo **polinomiale**.

## Preservazione delle dipendenze

Una decomposizione preserva le dipendenze se e solo se l'unione delle dipendenze indotte sui singoli schemi equivale alle dipendenze dello schema originale.

> La decomposizione $\rho = \{R_1(T_1), \dots, R_n(T_n)\}$ di $R(T, F)$ preserva le dipendenze se e solo se $\cup_i \pi_{T_i}(F) \equiv F$.

Per verificarlo applichiamo la definizione:

- Calcoliamo le proiezioni $\pi_{T_i}(F) = \{X \to Y \in F^+ | X \cap Y \subseteq T_i\}$
- Verifichiamo se $\cup_i \pi_{T_i}(F) \equiv F$.

### Verificare l'equivalenza

> $F \equiv G$ se e solo se $F \subseteq G^+ e G \subseteq F^+$

Sia $G = \cup_i \pi_{T_i}(F)$, per dimostrare che $F \equiv G$ osserviamo che:

- $F \subseteq G^+$ è verificabile tramite il problema dell'implicazione, perché equivale a verificare che per ogni $X \to Y \in F$ abbiamo $Y \subseteq X_G^+$
- $G \subseteq F^+$ vale per definizione, quindi non serve verificarlo

Ci manca quindi solo da calcolare $G = \cup_i \pi_{T_i}(F)$ per avere un algoritmo che verifica se le dipendenze sono preservate o meno.

### Calcolo delle proiezioni

⚠️ Esponenziale

```
function Project(F, Ti):
    proj = ∅
    for all X in Ti:
        Y = closure(X, F) - X
        proj = proj ∪ {X → Y ∩ Ti }
    return proj
```

### Ottimizzare la verifica

Esiste un algoritmo che calcola $X_G^+$ in tempo polinomiale senza calcolare G.

```
function FC(X, F, ρ):
    oldres = ∅
    newres = X
    while newres != oldres:
        oldres = newres
        for all Ri(Ti) ∈ ρ:
            newres = oldres ∪((newres ∩ Ti) + F ∩ Ti)
    return newres
```

Dati uno schema $R(T, F)$ e $\rho = R_1(T_1), \ldots, R_n(T_n)$ è dunque possibile verificare se $\rho$ preserva le dipendenze tramite il seguente algoritmo:

✅ Polinomiale

```
function PreserveDeps(R(T,F), ρ):
    for X -> Y in F:
        if Y !⊆ FC(X, F, ρ)
            return false
    return true
```

# Forme Normali

L'obiettivo delle forme normali è garantire che uno schema sia di buona qualità e viene spesso ottenuto tramite un processo di normalizzazione basato su una decomposizione dello schema di partenza.

> 1. Uno schema in forma normale non deve contenere anomalie
> 2. Il processo di normalizzazione deve preservare i dati
> 3. Il processo di normalizzazione deve preservare le dipendenze

Vedremo che in generale non possiamo garantire tutte e tre le proprietà!

## Boyce Codd Normal Form (BCNF)

> Uno schema di relazione $R(T, F)$ è in BCNF se e solo se per ogni dipendenza funzionale $X \to Y \in F^+$ tale che $Y \not\subseteq X$ si ha che $X$ è una superchiave.

Quindi: un database è in BCNF se, per ogni dipendenza funzionale $X \to Y \in F^+$ tale che $Y$ non sia un sottoinsieme di $X$, $X$ è una superchiave di $R$. Ciò implica che ogni dipendenza funzionale che coinvolge attributi non completamente determinati da una superchiave viola la BCNF.

Una dipendenza che viola la BCNF è detta **anomala**.

L'algoritmo di conversione in BCNF è detto algoritmo di **analisi**.

> Sia $R(T, F)$ lo schema di partenza:
>
> - Se $R(T, F)$ è già in BCNF, ritorna $\{R(T, F)\}$
> - Seleziona $X \to Y \in F$ che viola BCNF. Calcola gli insiemi di attributi $T_1 = X_F^+$ e $T_2 = X \cup (T - T_1)$
> - Calcola le proiezioni $F_1 = \pi_{T_1}(F)$ e $F2 = \pi_{T_2}(F)$
> - Decomponi ricorsivamente $R_1(T_1, F_1)$ e $R_2(T_2, F_2)$ in $\rho_1$ e $\rho_2$

> - Ritorna la loro unione $\rho_1 \cup \rho_2$

> **Pregi**
>
> - BCNF garantisce l'assenza di anomalie (no dipendenze anomale)
> - L'algoritmo di conversione in BCNF preserva i dati
> - Verificare se uno schema è in BCNF ha costo polinomiale

> **Difetti**
>
> - L'algoritmo di conversione in BCNF ha costo esponenziale, perch`e richiede di calcolare le proiezioni delle dipendenze
> - L'algoritmo di conversione in BCNF non preserva le dipendenze nel caso generale

# Terza Forma Normale (3NF)

> Uno schema di relazione $R(T, F)$ è in **3NF** se e solo se per ogni dipendenza funzionale $X \to Y \in F^+$ tale che $Y \not\subseteq X$ si ha che $X$ è una superchiave oppure tutti gli attributi di $Y - X$ sono primi.

Si può dimostrare che è possibile sostituire $F^+$ con $F$ nella definizione, ma verificare se uno schema è in 3NF ha comunque costo esponenziale, perché il calcolo degli attributi primi richiede di trovare tutte le chiavi.

> Per definizione ogni schema in BCNF è anche in 3NF, ma non viceversa.

Condizioni:

- Tutti gli attributi non chiave dello schema devono dipendere funzionalmente solo dalla chiave primaria (o da una superchiave) dello schema. In altre parole, se hai un attributo $A$ che dipende funzionalmente da un insieme di attributi $B$ che non sono una chiave, allora $A$ deve essere una chiave o una parte di una chiave.
- Non devono esserci dipendenze funzionali transitivamente dipendenti dalla chiave primaria dello schema. Ciò significa che se hai una dipendenza funzionale $X \to Y$ e una dipendenza funzionale $Y \to Z$, allora $Z$ deve essere una chiave o una parte di una chiave.

Algoritmo:

> Sia $R(T, F)$ lo schema di partenza:
>
> 1. Costruisci $G$, una copertura canonica di $F$
> 2. Sostituisci in $G$ ciascun insieme di dipendenze $X \to A_1, \ldots, X \to A_n$ con una singola dipendenza $X \to A_1 \ldots A_n$
> 3. Per ogni $X \to Y \in G$, crea un nuovo schema $S_i(XY)$
> 4. Elimina ogni schema contenuto in un altro schema
> 5. Se la decomposizione non contiene alcuno schema i cui attributi costituiscano una superchiave per $R$, aggiungi un nuovo schema $S(W)$ dove $W$ è una chiave di $R$

Esempio:

> Consideriamo $T = \{A, B, C, D, E\}$ e $F = \{AB \to C, C \to D, D \to E\}$
>
> 1. Applichiamo il passaggio 1 e otteniamo

$F^+ = \{AB \to C, AB \to D, AB \to E, C \to D, D \to E\}$

2. Sostituzione degli insiemi di dipendenze:
   $G = \{AB \to CDE, C \to D, D \to E\}$
3. Creiamo uno schema per ogni dipendenza in $G$:
   $S_1(ABCDE)$ - dalla dipendenza $AB \to CDE$
   $S_2(CD)$ - dalla dipendenza $C \to D$
   $S_3(DE)$ - dalla dipendenza $D \to E$
4. Eliminazione degli schemi contenuti in altri schemi:
   $S_2$ ed $S_3$ sono entrambi contenuti in $S_1$, quindi li eliminiamo
5. Verifichiamo se lo schema $S_1$ ha tutti gli attributi che costituiscono una superchiave per $R$. In questo caso $ABCDE$ è già una chiave per $R$, quindi non è necessario aggiungere altri schemi.

# Sicurezza nei Database

## Autenticazione e Autorizzazione

Tutti i DMBS implementano meccanismi di

- **autenticazione**
- **autorizzazione**

Il **controllo degli accessi** è il meccanismo con cui viene verificato che chi richiede un'operazione sia effettivamente autorizzato a farla.

In Postgres, il controllo dell'autenticazione è controllato dal file di configurazione `pg_hba.conf`.
È possibile specificare il metodo di autenticazione desiderato per ciascuna richiesta di autenticazione, definita da una quadrupla che include:

1. **tipo di connessione**: locale, remota, cifrata (via TLS)...
2. **database**: lista di database o keyword all
3. **utente**: lista di utenti o keyword all
4. **indirizzo**: hostname, indirizzo IP, range di IP...

Dopo la fase di autenticazione il DBMS sa con chi sta interagendo e pu`o implementare appropriate politiche di autorizzazione.

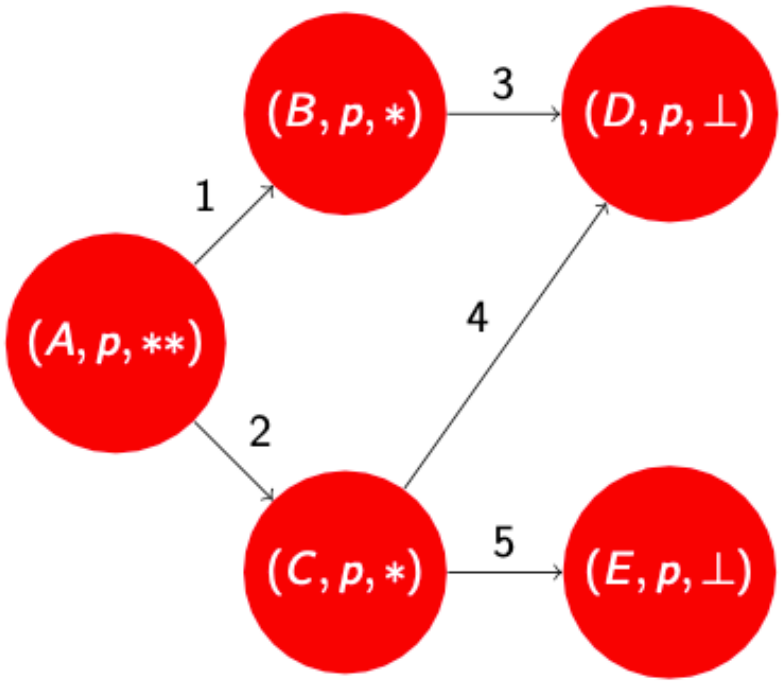
Regole base:

1. quando un oggetto (es. una tabella) viene creato, il suo creatore ne diventa il proprietario e può farne ciò che desidera
2. gli altri utenti invece possono accedere all'oggetto solamente con le modalità stabilite dai permessi concessi su di esso
3. il privilegio di eliminare un oggetto o alterarne la definizione è di pertinenza esclusiva del creatore dell'oggetto!

## Diagramma di autorizzazione

> Un diagramma di autorizzazione è un grafo orientato i cui nodi sono etichettati con una tripla (u, p, m), dove u è un utente, p è un permesso e m può avere una delle seguenti forme:
>
> 1. 1 $\perp$: il permesso è stato assegnato, ma non pu`ò essere delegato
> 2. 2 $*$: il permesso è stato assegnato e può essere delegato
> 3. 3 $**$: il permesso è stato concesso in qualità di proprietario

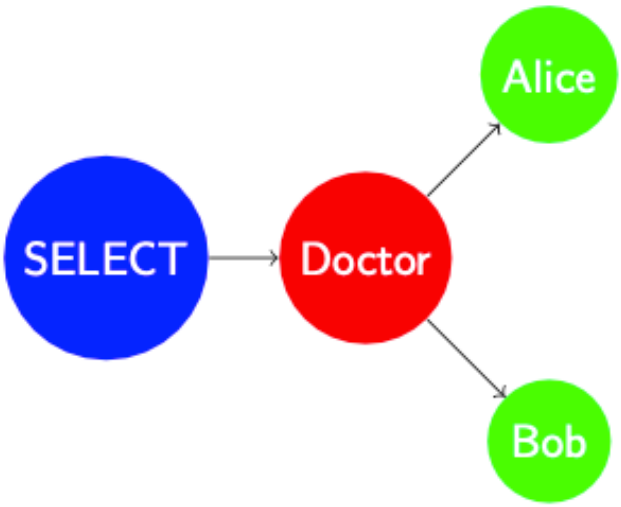
> Un arco da $(u_2, p, m_1)$ a $(u_2, p, m_2)$ modella che $u_1$ detiene il permesso $p$ con modalità $m_1$ e lo ha delegato ad $u_2$ con modalità $m_2$.




## Ruoli

Assegnare manualmente i permessi ad ogni singolo utente è spesso un processo costoso ed error-prone, visto l'elevato numero di utenti.

Un **ruolo** è un collettore di permessi, che permette di introdurre un livello di indirezione durante la loro assegnazione.




**Benefici**

I ruoli hanno numerosi benefici rispetto all'uso tradizionale dei permessi:

1. i ruoli raggruppano insiemi di permessi **logicamente collegati**
2. è molto meno costoso assegnare ruoli che permessi, visto che ci sono molti meno ruoli che permessi
3. è molto più difficile sbagliare l'assegnazione di un ruolo che di un insieme di permessi
4. le operazioni di revoca sono analogamente semplificate
5. i ruoli non sono attivi di default, contrariamente ai permessi: questo è più fedele al principio del minimo privilegio

**Ereditarietà**

Le opzioni `INHERIT` (default) e `NOINHERIT` consentono di gestire il meccanismo di ereditarietà:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

# SQL Injections

Consideriamo una web application che consenta di cercare le informazioni relative ad un utente, presentando il suo nome ($u) e password ($p).

```
user = get_parameter($u)
pass = get_parameter($p)
statement = "SELECT * FROM users WHERE name = '" +
             user + "' AND pwd = '" + pass + "';"
```

Il codice costruisce la query SQL da eseguire tramite concatenazione di stringhe, che per`o potrebbero essere state passate da un attaccante...

Se passiamo nome utente `marco` e password `' OR '1' = '1`

```
SELECT * FROM users WHERE name = 'marco' AND pwd = ''
                      OR '1' = '1';
```

Questa query ritorna l'intero contenuto della tabella users, andando a comprometterne la confidenzialità!

Ci sono due approcci tradizionali per prevenire SQL injection:

- **sanitizzazione**: analisi o trasformazione degli input processati per garantire l'assenza di contenuti malevoli, per esempio costrutti SQL
- **encoding**: trasformazione degli output generati per garantire che essi non vengano interpretati come codice eseguibile

L'escaping è una delle più semplici forme di encoding, che converte caratteri speciali nella loro versione "letterale"

```
userName  = escape(get_parameter($u))
pwd       = escape(get_parameter($p))
statement = "SELECT * FROM users WHERE name = '" +
             userName + "' AND password = '" + pwd + "';"
```

Un **prepared statement** è un'istruzione SQL contenente dei "buchi" detti parametri, che vengono riempiti in modo disciplinato (tipato).

```
userName  = get_parameter($u)
pwd       = get_parameter($p)
statement = "SELECT * FROM users WHERE name = ?
statement.setString(1,userName);
statement.setString(2,pwd);
```

Tuttavia, si consideri la seguente porzione di codice usata per stampare il contenuto di una tabella selezionabile da un menu` a tendina:

```
table = get_parameter($t)
statement = "SELECT * FROM " + table;
```

poichè la sintassi dei prepared statement non supporta l'uso di parametri nel nome della tabella, è importante sanitizzare l'input ricevuto.

```
table = get_parameter($t);
if (table == "Student" || table == "Teacher") {
  statement = "SELECT * FROM " + table;
}
else { throw new Exception("Unexpected!"); }
```

# Linguaggi

Molte applicazioni hanno bisogno di interfacciarsi con un database: in questa lezione studieremo il design space delle possibilit`a esplorate fino ad oggi e ne discuteremo pro e contro.

1. Linguaggi **integrati**: linguaggi che estendono SQL con tradizionali costrutti di programmazione general purpose (es. PL/pgSQL)
2. Linguaggi che **ospitano SQL**: linguaggi tradizionali, la cui sintassi viene estesa con costrutti SQL inframmezzati al codice
3. Utilizzo di **API**: linguaggi tradizionali, che si appoggiano a librerie di interfacciamento con SQL

## Linguaggi Integrati

Estendono SQL con costrutti di programmazione tradizionali.

```
CREATE FUNCTION cheapest() RETURNS integer AS $$
DECLARE r pc;
BEGIN
SELECT * INTO r FROM pc ORDER BY price;
RETURN r.price;
END; $$ LANGUAGE plpgsql;
```

**Vantaggi:**

- Stesso livello di astrazione di SQL
- Supporto per controlli statici da parte del type system

**Svantaggi:**

- Costo elevato di apprendimento: nuovo linguaggio
- Tecnologie proprietarie e non standardizzate
- Non adatti allo sviluppo di applicazioni complesse

## Linguaggi che ospitano SQL

Estendono un linguaggio tradizionale con costrutti SQL.

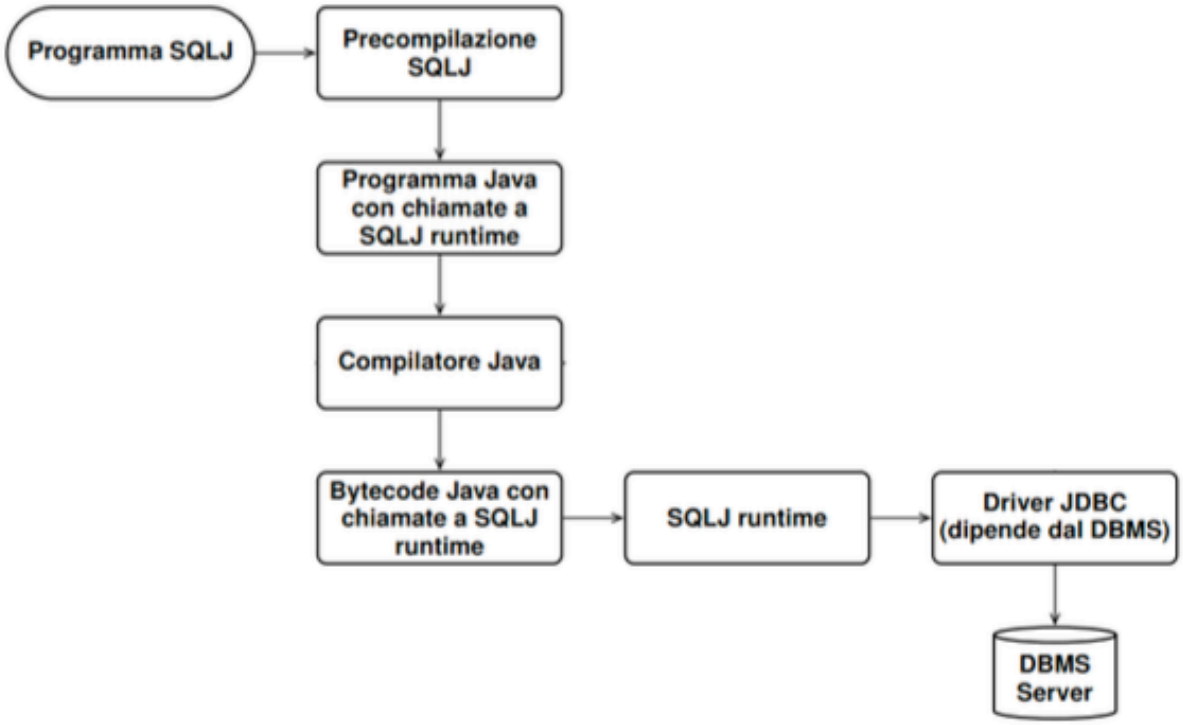
**Vantaggi:**

- Costo ridotto di apprendimento
- La sintassi SQL integrata abilita controlli statici (es. typing)

**Svantaggi:**

- Impedance mismatch: differenza di tipi fra linguaggio ospite e SQL
- È richiesto un preprocessore

**SQLJ (Deprecato)**




Una **connessione** definisce il contesto di interazione col DBMS, indicando il database acceduto e l'utente che vi si interfaccia (autenticazione).

```
Class.forName(DatabaseDriver); // carica il driver
#sql context MyContext;
MyContext contesto = new MyContext(urlDb, user, pwd);
#sql [contesto] INSERT INTO Province VALUES ('Venezia', 'VE', 44);
```

## API di interfacciamento a SQL

Non cambiano il linguaggio, appoggiandosi a librerie esterne.

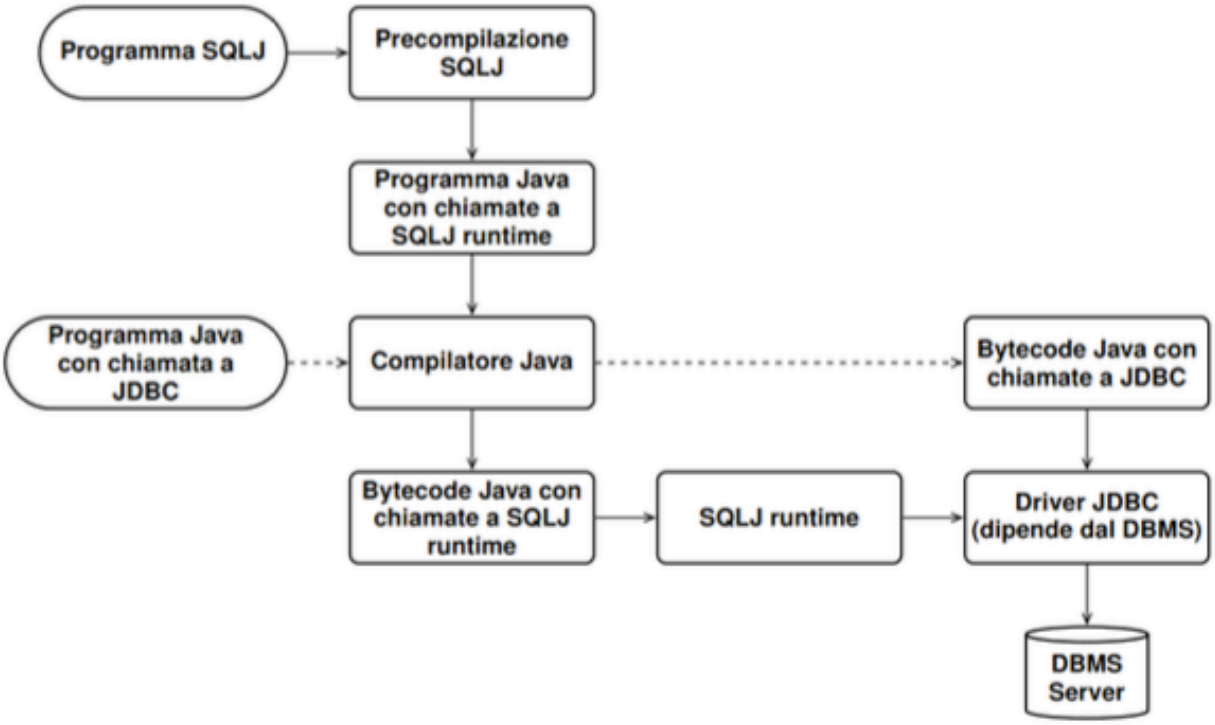
**Vantaggi:**

- Costo ridotto di apprendimento
- Non è richiesto alcun pre-processore
- Possibilità di utilizzo di SQL dinamico

**Svantaggi:**

- Impedance mismatch: differenza di tipo fra linguaggio ospite e SQL
- Assenza di controlli a tempo di compilazione

**JDBC**




Come SQLJ, anche JDBC si appoggia a connessioni per interagire con il database con cui vogliamo interfacciarci:

```
String db = "jdbc:oracle:oci";
String u  = "stefano";
String p  = "secret";
Connection con = DriverManager.getConnection(db, u, p);
```

Le query da eseguire sono passate alla libreria come **stringhe**, quindi potenzialmente calcolate a runtime e non soggette ad analisi statica.

```
Statement stmt = con.createStatement();
String val = "VALUES (Venezia, VE, 44)"
String query = "INSERT INTO Province " + val;
stmt.executeQuery(query);
```

Una soluzione più robusta si basa sull'uso di prepared statement, cioè comandi SQL con "buchi" da riempire:

```
String template = "INSERT INTO Province VALUES (?,?,?)";
PreparedStatement pstmt = con.prepareStatement(template);
pstmt.setString(1, "Venezia");
pstmt.setString(2, "VE");
pstmt.setInt(3, 44);
pstmt.executeQuery();
```

### ORM

L'ORM è una tecnica di programmazione che mitiga i problemi associati all'impedance mismatch, pur mantenendo un approccio basato su API.

JDBC

```
Statement stmt = con.createStatement();
String query = "SELECT * FROM Students WHERE id = 10"
ResultSet rs = stmt.executeQuery(query);
String name = rs.next().getString("Name");
```

ORM

```
Student stud = db.getStudent(10);
String name = stud.getName();
```

**Vantaggi:**

- Indipendenza dallo specifico DBMS sottostante
- Non richiede conoscenza approfondita di SQL
- Migliore supporto da parte del compilatore
- Astrazione da dettagli di basso livello (es. sanitizzazione delle query)

**Svantaggi:**

- Tipicamente più lento rispetto a SQL
- Poco adatto all'esecuzione di query complesse

# Indici

# NoSQL

NoSQL (Not Only SQL) si riferisce ad un insieme di database non relazionali che offrono un'alternativa ai tradizionali database relazionali. A differenza dei database relazionali, che organizzano i dati in tabelle strutturate e utilizzano SQL per interrogare e maniplare i dati, i database NoSQL adottano approccit diversi.

I database NoSQL sono progettati per gestire grandi volumi di dati non strutturati, semi-strutturati o dati che cambiano frequentemente la loro struttura. Sono spesso usati in scenari in cui la scalabilità orizzontale e la disponibilità dei dati sono prioritari rispetto alla coerenza assoluta dei dati.

Esistono:

1. Database **document-oriented**: memorizzano i dati in documenti JSON, XML o altri formati simili. Questi documenti sono autocontenuti e possono rappresentare strutture di dati complesse. Esempi noti sono MongoDB e CouchDB.
2. Database di tipo **key-value**: memorizzano i dati in coppia chiave-valore. La chiave viene utilizzata per

recuperare rapidamente il valore corrispondente. Esempi famosi sono Redis e Riak.
3. Database **a colonne**: memorizzano i dati in colonne invece di righe, consentendo una rapida lettura selettiva dei dati. Esempi di database a colonne includono Cassandra e HBase.
4. Database **a grafo**: memorizzano i dati come grafi di nodi e archi, consentendo una rappresentazione e una query efficienti delle relazioni tra gli oggetti. Esempi di database a grafo includono Neo4j e OrientDB.

## Neo4j

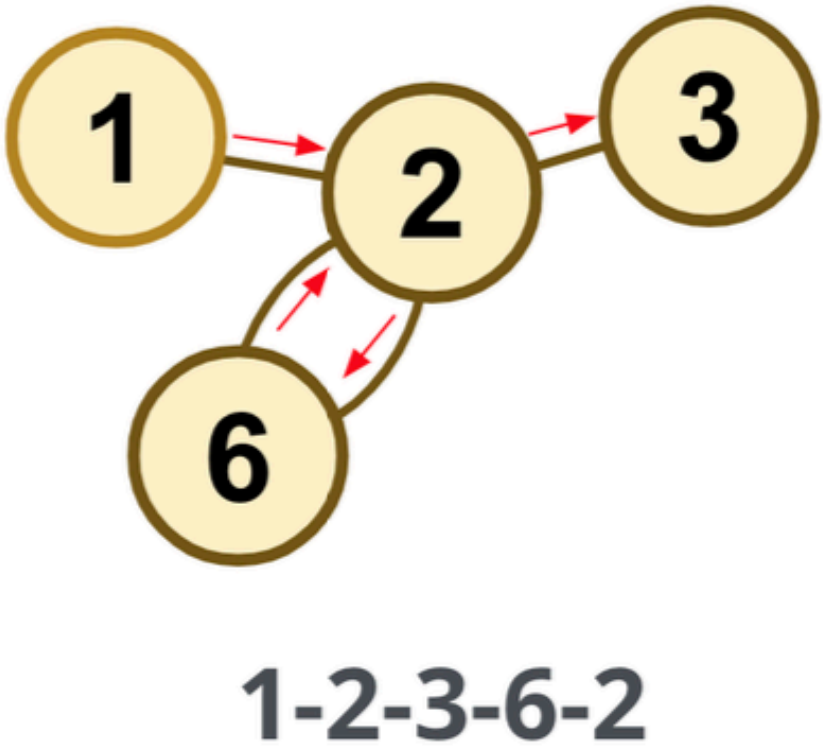
Neo4j è un datbase a grafo, noto anche come sistema di gestione di database a grafo (Graph Database Management System).

In Neo4j i dati sono organizzati in:

- nodi: rappresentano entità o oggetti
- Relazioni: descrivono le connessioni o le interazioni tra i nodi

Ogni nodo può avere un insieme di proprietà che rappresentano i suoi attributi. Le relazioni, invece, hanno una direzione, un tipo e possono avere anche proprietà.

In ogni attraversamento di un grafo, ogni nodo può essere visitato più di una volta, ma ogni relazione è attraversata al massimo una volta. Ciò è chiamato un sentiero (path).




### DBMS

Nel DBMS Neo4j abbiamo:

- un database **system** che meorizza metadati dei database d'installazione e configurazioni di sicurezza
- un database **default** (chiamato di default neo4j) che è effettivamente il database utilizzato dall'utente in cui viene salvato il modello dati a grafo

In Neo4j Enterprise ci possono essere più di un database utente.

**Indirizzamente senza indicizzazione**

Capacità di accedere ai nodi e alle relazioni in un grafo direttamente e in modo efficiente, senza dover fare affidamento su indici o ricerche elaborate.

Nei database relazionali tradizionali, l'accesso ai dati avviene generalmente tramite indici che puntano alle posizioni fisiche dei dati sul disco. Tuttavia, nei database a grafo, ogni nodo è collegato direttamente ai suoi vicini tramite le relazioni. Ciò significa che, per accedere a un nodo e alle sue relazioni adiacenti, non è necessario eseguire una ricerca complessa attraverso un indice, ma è sufficiente seguire direttamente i collegamenti nel grafo.

**ACID**

ACID è un acronimo che sta per Atomicità (Atomicity), Coerenza (Consistency), Isolamento (Isolation) e Durabilità (Durability). Questi concetti rappresentano le proprietà fondamentali che un sistema di gestione di database dovrebbe garantire per garantire l'integrità e l'affidabilità dei dati.

- **Atomicità**: garantisce che un'operazione di database sia considerata una singola unità indivisibile. Se un'operazione coinvolge più azioni o modifiche ai dati, viene garantito che tutte le azioni vengano eseguite correttamente o che nessuna di esse venga eseguita affatto. Se una parte dell'operazione non può essere completata, l'intera operazione viene annullata e i dati vengono ripristinati allo stato precedente, mantenendo il database in uno stato coerente
- **Coerenza**:  garantisce che il database si trovi in uno stato valido prima e dopo l'esecuzione di un'operazione. Le operazioni devono rispettare un insieme di regole o vincoli predefiniti per mantenere l'integrità dei dati
- **Isolamento**: L'isolamento garantisce che l'esecuzione simultanea di più transazioni non interferisca l'una con l'altra. Ogni transazione deve essere eseguita come se fosse l'unica transazione in corso nel sistema, garantendo che i risultati delle transazioni concorrenti siano coerenti e prevedibili
- **Durabilità**: La durabilità assicura che una volta che un'operazione di scrittura è stata completata con successo, i risultati siano permanenti e resistenti a guasti o riavvii del sistema. I dati scritti sul disco o su un supporto di memorizzazione persistente devono rimanere disponibili anche in caso di interruzioni di corrente o altri eventi catastrofici.

## Cypher

Una delle caratteristiche distintive di Neo4j è la sua architettura nativa a grafo, che consente prestazioni elevate nelle operazioni di interrogazione e analisi dei dati interconnessi. Le query in Neo4j vengono eseguite utilizzando il linguaggio di interrogazione Cypher, che è specificamente progettato per esprimere pattern di grafo e operazioni di navigazione nel grafo in modo intuitivo.

In SQL:

```
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"
```

mentre in Cypher:

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

# Redis

**RE**mote **DI**ctionary **S**erver

Redis è un sistema di memorizzazione dei dati ad altre prestazioni, noto come data store in memoria. È un database NoSQL open-source di tipo chiave valore, in cui i dati vengono archiviati in memoria principale per consentire un accesso veloce e ad alte prestazioni.

Redis è progettato per essere leggero e veloce, ed è spesso utilizzato come cache o come archivio di dati temporanei per migliorare le prestazioni delle applicazioni. È in grado di gestire grandi quantità di dati in modo efficiente, fornendo accesso ai dati tramite un'ampia gamma di strutture dati, come stringhe, liste, set, hash e ordini di classificazione.

Una delle caratteristiche più interessanti di Redis è il supporto alle operazioni atomiche sui dati. Ciò significa che Redis garantisce che le operazioni di lettura/scrittura su una singola chiave siano eseguite in modo atomico, senza interferenze da altre operazioni. Questa caratteristica consente la gestione di strutture dati complesse e la sincronizzazione tra thread o processi.

## Chiavi

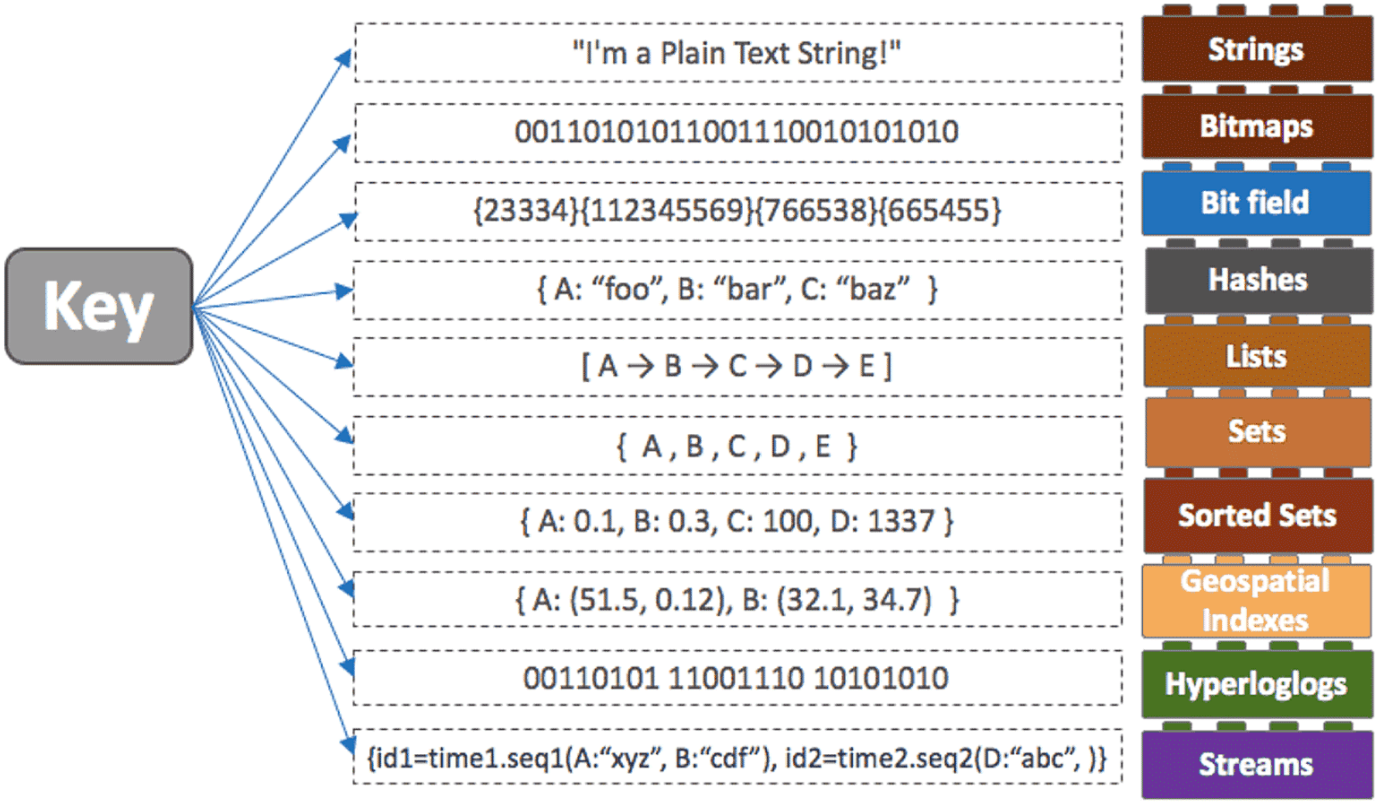
In redis un record viene chiamato chiave-valore. Ogni chiave in redis è associata a un valore. La chiave è una stringa che non può essere più grane di 512MB.
Esempi:

- ❌ `user1000:id1000:followersOfUser1000`
- ❌ `u1000flw`
- ✅ `user:100:followers`

Operazioni sulle chiavi:

- **KEYS**: Restituisce tutte le chiavi corrispondenti a un determinato pattern.
- **EXISTS**: Verifica se una chiave esiste.
- **DEL**: Rimuove una chiave e il suo valore associato.
- **(P)EXPIRE**: Imposta un tempo di scadenza per una chiave, dopo il quale verrà automaticamente eliminata.
- **TTL**: Restituisce il tempo rimanente di scadenza di una chiave.
- **PERSIST**: Rimuove il tempo di scadenza di una chiave, rendendola persistente.
- **TYPE**: Restituisce il tipo di dato associato a una chiave (stringa, lista, set, hash, sorted set, ecc.).
- **RENAME**: Rinomina una chiave.
- **SCAN**: Esegue una scansione di tutte le chiavi nel database in modo iterativo.
- **RANDOMKEY**: Restituisce una chiave casuale dal database.

Mentre una chiave è una stringa, un valore può essere di diversi tipi:




## Valori

### Stringhe

Un valore può essere una semplice stringa di testo. Le stringhe in Redis possono rappresentare qualsiasi tipo di dato, come numeri interi, numeri in virgola mobile o dati serializzati.

Operazioni su Stringhe:

- **GET**: Recupera il valore associato a una chiave.
- **SET**: Imposta il valore di una chiave.
    - Le stringhe possono anche essere usate come **contatori**:
        - **INCR(BY)/DECR(BY)**: Incrementa o decrementa il valore numerico di una chiave.
- **APPEND**: Aggiunge del testo a una stringa esistente.
- **STRLEN**: Restituisce la lunghezza di una stringa.
- **MGET/MSET**: Recupera o imposta i valori di più chiavi in un'unica operazione.

### Liste

Una lista è una sequenza ordinata di elementi. I valori possono essere aggiunti o rimossi dalla lista in posizioni specifiche.

Operazioni su Liste

- **LPUSH/RPUSH**: Aggiunge un elemento all'inizio o alla fine di una lista.
- **LPOP/RPOP**: Rimuove e restituisce l'elemento all'inizio o alla fine di una lista.
- **LLEN**: Restituisce la lunghezza di una lista.

- **LRANGE**: Restituisce un intervallo di elementi di una lista.
- **LINSERT**: Inserisce un elemento in una lista prima o dopo un elemento esistente.

**Hash**

Un hash è una struttura dati composta da campi associati a valori. Gli hash in Redis sono spesso utilizzati per rappresentare oggetti o record, in cui ogni campo rappresenta un attributo dell'oggetto.

Operazioni su Hash

- **HSET/HGET**: Imposta o recupera il valore di un campo in un hash.
- **HMSET/HMGET**: Imposta o recupera i valori di più campi in un hash.
- **HDEL**: Rimuove uno o più campi da un hash.
- **HGETALL**: Restituisce tutti i campi e i valori di un hash.
- **HINCRBY**: Incrementa il valore numerico di un campo in un hash.

**Set**

Un set è una raccolta non ordinata di elementi unici. I set in Redis forniscono operazioni come l'aggiunta, la rimozione e la verifica dell'appartenenza di un elemento.

Operazioni su Set

- **SADD**: Aggiunge uno o più elementi a un set.
- **SREM**: Rimuove uno o più elementi da un set.
- **SISMEMBER**: Verifica se un elemento è presente in un set.
- **SMEMBERS**: Restituisce tutti gli elementi di un set.
- **SUNION/SINTER/SDIFF**: Esegue operazioni di unione, intersezione o differenza tra set.

**Sorted Set**

Un sorted set è una raccolta di elementi ordinati in base a un punteggio associato. Gli elementi possono essere recuperati in base al punteggio o eseguire operazioni come l'aggiunta, la rimozione o l'aggiornamento del punteggio.

- **ZADD**: Aggiunge un elemento con un punteggio a un sorted set.
- **ZREM**: Rimuove un elemento da un sorted set.
- **ZRANK/ZSCORE**: Restituisce il rango o il punteggio di un elemento in un sorted set.
- **ZRANGE/ZREVRANGE**: Restituisce un intervallo di elementi in base al punteggio da un sorted set.
- **ZUNIONSTORE/ZINTERSTORE**: Esegue l'unione o l'intersezione di uno o più sorted set e salva il risultato in un nuovo sorted set.

**Bitmap**

Un bitmap in Redis è una struttura dati che rappresenta una sequenza di bit, dove ogni bit può essere impostato su 0 o 1. Redis offre operazioni efficienti per manipolare i bitmap, come impostare o ottenere il valore di un singolo bit, contare il numero di bit impostati a 1 e operazioni logiche come l'AND, l'OR e lo XOR tra bitmap. I bitmap in Redis sono spesso utilizzati per il tracciamento di flag o per eseguire operazioni di filtraggio o conteggio efficienti.

**Geospatial**

Il geospatial index in Redis è una struttura dati che consente di memorizzare e interrogare dati geografici. Redis supporta l'indicizzazione spaziale tramite un sistema di coordinate geografiche, consentendo di memorizzare punti geografici e di eseguire operazioni come la ricerca di elementi entro una determinata distanza o il calcolo della distanza tra due punti. Ciò rende Redis adatto per applicazioni che richiedono la gestione di dati geografici, come la ricerca di luoghi nelle vicinanze o la creazione di servizi di localizzazione.

**HyperLogLog**

L'HyperLogLog (HLL) in Redis è una struttura dati probabilistica che consente di stimare la cardinalità (il numero di elementi unici) di un insieme di dati molto grande. HLL consente di ottenere una stima approssimata della cardinalità con un consumo di memoria molto ridotto rispetto all'archiviazione dei dati effettivi. Questa struttura dati è utile in situazioni in cui è necessario stimare rapidamente la distinzione approssimata di un grande insieme di dati, come il conteggio delle visualizzazioni di pagina uniche o l'identificazione di utenti unici.

**Streams**

Gli Streams in Redis sono una struttura dati di log che rappresenta un flusso di eventi o messaggi ordinati in base al tempo. Gli stream consentono di aggiungere nuovi eventi alla fine del flusso e recuperarli in base all'ordine temporale o tramite operazioni di lettura complesse come il raggruppamento, il filtraggio e l'elaborazione dei dati. Le caratteristiche degli stream rendono Redis un'opzione interessante per la gestione di dati in tempo reale, come la coda dei messaggi, l'elaborazione di eventi o la gestione dei log.