Database: Algebra e Calcolo Relazionale

Basato sulle slide di Danilo Montesi

15 maggio 2025

1 Introduzione: Linguaggi per Basi di Dati

1.1 Tipi di Operazioni

I linguaggi per basi di dati si dividono in base al tipo di operazioni che permettono:

- **Operations on the schema** (Operazioni sullo schema):
  - **DDL (Data Definition Language):** Per definire la struttura del database (tabelle, vincoli, ecc.).
- **Operations on data** (Operazioni sui dati):
  - **DML (Data Manipulation Language):** Per manipolare i dati (inserire, modificare, cancellare, interrogare).
    - Query instructions (Istruzioni di interrogazione): Per estrarre i dati di interesse (`SELECT` in SQL).
    - Update instructions (Istruzioni di aggiornamento): Per inserire nuovi dati, modificare o cancellare dati esistenti (`INSERT`, `UPDATE`, `DELETE` in SQL).

1.2 Query Languages per Relational DBs

Le query languages possono essere classificate in base a come specificano il risultato:

- **Declarative** (Dichiarative): Specificano **cosa** (le proprietà) dei risultati che vogliamo ottenere. Il sistema decide come ottenerli.
- **Imperative/Procedural** (Imperative/Procedurali): Specificano **come** il risultato viene ottenuto (una sequenza di operazioni).

Esempi:

- **Relational Algebra:** Procedurale. Teorica, non implementata direttamente come linguaggio di query per gli utenti finali, ma alla base dei query processor dei DBMS.
- **Relational Calculus:** Dichiarativo. Teorico, non implementato direttamente per gli utenti finali.
- **SQL (Structured Query Language):** Parzialmente dichiarativo. Lo standard implementato e più diffuso.
- **QBE (Query by Example):** Dichiarativo. Implementato (es. in Microsoft Access).

2 Algebra Relazionale

L'Algebra Relazionale è un linguaggio di query procedurale definito da un insieme di operatori che operano su relazioni e producono relazioni come risultato. Gli operatori possono essere combinati (composizionali).

2.1 Operatori Fondamentali

- Set Operators (operazioni insiemistiche): **union ($\cup$), intersection ($\cap$), difference ($-$)**. Trattano le relazioni come insiemi di tuple. Richiedono che le relazioni abbiano lo stesso schema (compatibilità di unione).
- **Rename (ρ)**: Operatore unario che cambia i nomi degli attributi o della relazione stessa.
- **Select (σ)**: Operatore unario che seleziona un sottoinsieme di tuple in base a un predicato.
- **Project (π)**: Operatore unario che seleziona un sottoinsieme di attributi (colonne) e rimuove i duplicati.
- **Join ($\bowtie, \times, \bowtie_{...}(...)$)**: Operatori binari che combinano tuple da due relazioni.

2.2 Set Operators

Le relazioni sono viste come insiemi di tuple. Per poter applicare unione, intersezione o differenza, le relazioni devono avere lo **stesso schema** (stesso numero di attributi, stessi nomi e stessi domini, oppure nomi diversi ma resi compatibili con Rename).

Esempi dalle slide: GRADUATED(Number, Name, Age), TECHNICIANS(Number, Name, Age). Schema compatibile, quindi operazioni insiemistiche dirette:

- $\texttt{GRADUATED} \cup \texttt{TECHNICIANS}$: Tutte le tuple che sono in GRADUATED o in TECHNICIANS (o in entrambe).
- $\texttt{GRADUATED} \cap \texttt{TECHNICIANS}$: Tutte le tuple che sono sia in GRADUATED che in TECHNICIANS.
- $\texttt{GRADUATED} - \texttt{TECHNICIANS}$: Tutte le tuple che sono in GRADUATED ma non in TECHNICIANS.

Se avessimo FATHERHOOD(Father, Child) e MOTHERHOOD(Mother, Child), per fare l'unione, dovremmo rinominare: $\rho_{\text{Father} \leftarrow \text{Mother}}(MOTHERHOOD) \cup$ FATHERHOOD.

2.3 Renaming (ρ)

- Operatore **unario**.
- Usato per **cambiare il nome degli attributi** o della relazione stessa. Lo schema cambia, i dati rimangono uguali.
- Sintassi per attributi: $\rho_{NewName \leftarrow OldName}(RELATION)$. Esempio: $\rho_{Parent \leftarrow Father}(FATHERHOOD)$ crea una nuova relazione identica a FATHERHOOD ma con l'attributo 'Father' rinominato 'Parent'.
- Sintassi per relazione: $\rho_{NewRelationName}(RELATION)$.

Il renaming è fondamentale per rendere compatibili gli schemi per le operazioni insiemistiche o per distinguere attributi con lo stesso nome provenienti da relazioni diverse in un join (anche se il natural join fa questo implicitamente, è esplicito in theta-join/equi-join notazionali come visto nelle slide).

2.4 Selection (σ)

- Operatore **unario**.
- Restituisce le tuple (righe) che soddisfano un **predicato** (condizione booleana).
- Lo schema del risultato è lo **stesso** dello schema di input.
- Sintassi: $\sigma_{Predicate}(RELATION)$.
- Predicate: espressione booleana sugli attributi della relazione (es. $Salary > 50$, $Office =' Milan'$, $Surname = Office$).
- Corrisponde alla clausola `WHERE` in SQL:

```sql
SELECT * FROM Employee WHERE Salary > 50;
-- Corrisponde a: sigma_{Salary > 50}(EMPLOYEE)
```

2.5 Projection (π)

- Operatore **unario**.
- Restituisce una relazione con solo gli **attributi specificati** (colonne).
- Le **tuple duplicate vengono rimosse** nel risultato (poiché le relazioni sono insiemi).
- Lo schema del risultato è un **sottoinsieme** dello schema di input.
- Sintassi: $\pi_{AttributeList}(RELATION)$.
- Corrisponde alla clausola `SELECT` (senza `DISTINCT`) in SQL:

```sql
SELECT Number, Surname FROM Employee;
-- Corrisponde a: pi_{Number, Surname}(EMPLOYEE)
```

- **Cardinalità:** La cardinalità dell'output è **al massimo** quella dell'input. Può essere minore se la proiezione su un sottoinsieme di attributi causa duplicati che vengono rimossi. È uguale alla cardinalità dell'input se AttributeList include una superchiave della relazione.

2.6 Combinazione di Selection e Projection

- Selection filtra **orizzontalmente** (righe).
- Projection filtra **verticalmente** (colonne).
- Combinandoli, si estraggono tuple specifiche e solo i loro attributi desiderati: $\pi_{AttributeList}(\sigma_{Predicate}(RELATION))$.
- Esempio: Restituire numero e cognome degli impiegati con stipendio ¿ 50:

$$\pi_{\text{Number, Surname}}(\sigma_{\text{Salary ¿ 50}}(\text{EMPLOYEE}))$$

Corrisponde a:

```sql
SELECT Number, Surname
FROM Employee
WHERE Salary > 50;
```

Limiti: Questi operatori da soli permettono di estrarre informazioni solo da una **singola relazione**. Non permettono di correlare dati tra relazioni diverse o tuple diverse della stessa relazione in modo complesso.

2.7 Join

- Operatore **binario**.
- Permette di **combinare tuple** da due relazioni diverse in base a una condizione.
- È essenziale per correlare informazioni distribuite su più tabelle, come si fa con `JOIN` in SQL.

2.7.1 Cartesian Product ($\times$)

- Operatore binario.
- Combina **ogni tupla** della prima relazione con **ogni tupla** della seconda.
- Sintassi: $R_1 \times R_2$.
- Lo schema del risultato è l'unione degli attributi di R_1 e R_2. Se ci sono attributi con lo stesso nome, vengono qualificati (es. $R1.Attr$, $R2.Attr$).
- La cardinalità del risultato è $|R_1| \times |R_2|$.
- È raramente usato da solo in pratica; è più utile quando seguito da una Selection per filtrare le combinazioni desiderate.

2.7.2 Theta-Join ($\bowtie_{Condition}$ (...))

- Operatore binario definito come una Selection sul Cartesian Product: $R_1 \bowtie_{Condition} (R_2) = \sigma_{Condition}(R_1 \times R_2)$.
- Permette di specificare una **condizione arbitraria** (Condition, tipicamente con operatori $=, <, >, \leq, \geq, \neq$) per combinare le tuple.
- Sintassi: $R_1 \bowtie_C (R_2)$, dove C è la condizione.

2.7.3 Equi-Join

- Un caso speciale di Theta-Join in cui la condizione C è una **congiunzione di uguaglianze** ($=$).
- È molto comune ed efficiente. Permette di combinare tuple quando i valori di specifici attributi (anche con nomi diversi) sono uguali.
- Esempio: Combinare EMPLOYEE e DEPARTMENT basandosi sull'uguaglianza tra EMPLOYEE.Dept e DEPARTMENT.Code:

$$\text{EMPLOYEE} \bowtie_{\text{EMPLOYEE.Dept = DEPARTMENT.Code}} (\text{DEPARTMENT})$$

  Corrisponde a un `JOIN ON` in SQL:

```sql
SELECT *
FROM Employee JOIN Department
ON Employee.Dept = Department.Code;
```

2.7.4 Natural Join ($\bowtie$)

- Operatore binario.
- Combina tuple basandosi sull'**uguaglianza di valore su tutti gli attributi con lo stesso nome**.
- Lo schema del risultato è l'unione degli attributi di entrambe le relazioni, con gli attributi in comune che appaiono una sola volta.
- Sintassi: $R_1 \bowtie R_2$.
- Può essere definito in termini di Theta-Join e Projection. Esempio: se EMPLOYEE ha (Number, Name, Dept) e DEPARTMENT ha (Dept, Chief), EMPLOYEE $\bowtie$ DEPARTMENT combina su 'Dept' e il risultato avrà schema (Number, Name, Dept, Chief).
- Corrisponde, in molti casi, a `JOIN USING` o `NATURAL JOIN` in SQL, ma è importante essere precisi perché SQL ha sottili differenze o preferisce la notazione esplicita con `ON`.

```sql
SELECT *
FROM Employee NATURAL JOIN Department; -- Joins on 'Dept'
-- Equivalente (spesso preferito per chiarezza):
SELECT *
FROM Employee JOIN Department USING (Dept);
```

2.7.5 Cardinalità del Risultato del Join

- Il numero di tuple nel risultato di un join $R_1 \bowtie R_2$ (o $R_1 \bowtie_C (R_2)$) è compreso tra 0 e $|R_1| \times |R_2|$.
- Se l'attributo di join è una **chiave (PK) in** R_2, allora ogni tupla di R_1 può trovare al massimo una corrispondenza in R_2 (se esiste). La cardinalità è tra 0 e $|R_1|$.
- Se l'attributo di join è una **chiave (PK) in** R_2 e c'è un **vincolo di integrità referenziale** (Foreign Key in R_1 che referenzia la PK in R_2, non NULL), allora ogni tupla di R_1 troverà esattamente una corrispondenza in R_2. La cardinalità è esattamente $|R_1|$.

2.7.6 Inner Join vs Outer Join

- Il Join standard (Natural, Theta, Equi) è un **Inner Join**. Discarda le tuple che non trovano corrispondenza nell'altra relazione ("left out").
- Gli **Outer Join** mantengono le tuple che non trovano corrispondenza, riempiendo con valori **NULL** per gli attributi della relazione mancante.
- Tipi di Outer Join:
  - **Left Outer Join ($\bowtie\!\!-$):** Mantiene tutte le tuple della relazione **sinistra**. Corrisponde a `LEFT JOIN` o `LEFT OUTER JOIN` in SQL.
  - **Right Outer Join ($-\!\!\bowtie$):** Mantiene tutte le tuple della relazione **destra**. Corrisponde a `RIGHT JOIN` o `RIGHT OUTER JOIN` in SQL.
  - **Full Outer Join ($-\!\!\bowtie\!\!-$):** Mantiene tutte le tuple da **entrambe** le relazioni. Corrisponde a `FULL JOIN` o `FULL OUTER JOIN` in SQL.

2.8 Espressioni Equivalenti in Algebra Relazionale

Due espressioni di Algebra Relazionale sono equivalenti se producono lo stesso risultato su qualsiasi istanza valida del database (lo stato del database può cambiare, ma l'equivalenza deve valere sempre). Le regole di equivalenza sono cruciali per l'**ottimizzazione delle query** nei DBMS. Il query optimizer riscrive le query in forme equivalenti ma più efficienti.

Esempi di regole di equivalenza (quelle più importanti per l'efficienza):

- **Pushing selections down:** È spesso più efficiente applicare le selezioni **il prima possibile** per ridurre la dimensione delle relazioni intermedie.

$$\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$$

$$\sigma_C(R_1 \times R_2) = \sigma_C(R_1) \times R_2 \quad (\text{se } C \text{ solo su attr di } R_1)$$

$$\sigma_C(R_1 \times R_2) = R_1 \times \sigma_C(R_2) \quad (\text{se } C \text{ solo su attr di } R_2)$$

$$\sigma_{C_1 \wedge C_2}(R_1 \times R_2) = \sigma_{C_1}(R_1) \times \sigma_{C_2}(R_2) \quad (\text{se } C_1 \text{ solo su } R_1, C_2 \text{ solo su } R_2)$$

  Questo è il motivo per cui una query SQL con `WHERE` dopo un `FROM` con più tabelle è spesso più efficiente di un `SELECT * FROM R1, R2 WHERE ...`, perché il DBMS può "spingere" le condizioni `WHERE` (selection) verso le tabelle individuali o inserirle direttamente nel JOIN.

- **Pushing projections down:** È spesso più efficiente applicare le proiezioni **il prima possibile** per ridurre la dimensione (larghezza) delle relazioni intermedie. Bisogna fare attenzione a non rimuovere attributi necessari per operazioni successive (es. join o selection).

$$\pi_X(R_1 \times R_2) = \pi_{X_1}(R_1) \times \pi_{X_2}(R_2) \quad (\text{dove } X = X_1 \cup X_2, X_1 \subseteq \text{schema}(R_1), X_2 \subseteq \text{schema}(R_2))$$

  Per i join, bisogna proiettare solo dopo il join, ma si possono proiettare *prima* gli attributi che non servono *né per la condizione di join né per le proiezioni finali*.

$$\pi_{X_1 \cup Y_2}(R_1(X_1 \cup Y_1) \bowtie R_2(X_2 \cup Y_2)) = \pi_{X_1 \cup Y_1}(R_1) \bowtie \pi_{X_2 \cup Y_2}(R_2)$$

  (Questa è una semplificazione; le regole complete considerano gli attributi usati nella condizione di join e quelli proiettati alla fine). L'idea è rimuovere colonne non necessarie il prima possibile.

- La **proiezione non è distributiva sulla differenza**. $\pi_X(R_1 - R_2) \neq \pi_X(R_1) - \pi_X(R_2)$.
- Molte proprietà insiemistiche (commutatività, associatività) valgono per join, unione, intersezione, ma **non per la differenza**.

Nota importante: Non è lo scopo del corso imparare a riscrivere manualmente le query per l'efficienza in Algebra Relazionale. I moderni DBMS lo fanno automaticamente tramite il query optimizer. L'obiettivo è capire i concetti e le trasformazioni possibili.

2.9 Selection con valori NULL

- In presenza di valori `NULL`, le espressioni booleane possono avere tre valori: **TRUE**, **FALSE**, **UNKNOWN**.
- La Selection (σ) restituisce solo le tuple per cui il predicato è **TRUE**. Le tuple per cui è FALSE o UNKNOWN vengono scartate.
- Questo significa che un predicato come $\sigma_{Age>30ANDAge<=30}(PEOPLE)$ non restituirà tutte le persone se alcune hanno Age = NULL, perché per queste tuple la condizione $Age > 30ANDAge <= 30$ sarà UNKNOWN.
- Per gestire i NULL nelle selezioni, si usano predicati speciali come `IS NULL` e `IS NOT NULL`.
- Esempio: Selezionare le persone con Age maggiore di 40 **o** con Age NULL:

$$\sigma_{\text{Age ¿ 40 OR Age IS NULL}}(\text{PEOPLE})$$

In SQL:

```sql
SELECT *
FROM People
WHERE Age > 40 OR Age IS NULL;
```

Teoricamente si potrebbe usare una logica a 3 valori, ma nella pratica standard dei database si considera UNKNOWN come ”non TRUE” ai fini della selezione.

2.10 Views

- Una View è una **tabella virtuale** definita da una query. Non contiene dati propri; mostra i dati delle tabelle base sottostanti.
- Utili per fornire **rappresentazioni diverse** dei dati o **sottoschemi personalizzati** (Schema Esterno nell’architettura ANSI/SPARC).
- Permettono agli utenti di vedere solo i dati di loro interesse e con la struttura desiderata, nascondendo la complessità delle tabelle base.
- Possono anche servire come **strumento di programmazione** per semplificare query complesse suddividendole o riusando definizioni.

2.10.1 Tipi di Views

- **Virtual Relations (Views):** La definizione comune. La query sulla view viene **riscritta** dal DBMS in una query sulle tabelle base sottostanti. Non c’è storage separato per la view. Ampiamente supportate. Non migliorano l’efficienza dell’esecuzione (la query riscritta viene ottimizzata).
- **Materialized Views:** La query che definisce la view viene eseguita e il risultato viene **memorizzato fisicamente**. Forniscono accesso più veloce (lettura) ma occupano spazio e devono essere mantenute aggiornate quando le tabelle base cambiano (aggiornamenti lenti). Gli aggiornamenti *sulla* materialized view sono raramente supportati direttamente.

2.10.2 Esempio di View (Algebra Relazionale)

Definiamo una view `SUPERVISOR` che combina AFFILIATION e MANAGEMENT:

- `AFFILIATION(Employee, Dept)`
- `MANAGEMENT(Dept, Chief)`

La view `SUPERVISOR` potrebbe mostrare quale impiegato è supervisionato da quale Chief, unendo le due tabelle sul dipartimento:

$$\text{SUPERVISOR} := \pi_{\text{Employee, Chief}}(()\text{AFFILIATION} \bowtie \text{MANAGEMENT})$$

(Assumendo che gli attributi di join abbiano lo stesso nome o siano gestiti implicitamente dal natural join, altrimenti si userebbe un equi-join esplicito dopo un eventuale rename).

Query sulla view: Trova gli impiegati supervisionati da 'Mori'.

$$\sigma_{\text{Chief='Mori'}}(\text{SUPERVISOR})$$

Il DBMS riscrive questa query espandendo la definizione della view:

$$\sigma_{\text{Chief='Mori'}}((\pi_{\text{Employee, Chief}}(()\text{AFFILIATION} \bowtie \text{MANAGEMENT})))$$

Questo è l'query processing di base per le virtual views.

2.10.3 Aggiornare le Views (Incremental Updates)

Aggiornare una view significa modificare i dati nelle tabelle base in modo che la view aggiornata rifletta la modifica.

- Per ogni aggiornamento sulla view deve esserci una **corrispondente (e possibilmente unica) modifica sulle tabelle base**.
- Non è sempre univoco capire quali tuple delle tabelle base modificare per riflettere un cambiamento nella view.
- Quindi, solo un **sottoinsieme limitato** di views sono aggiornabili (tipicamente views semplici definite su una singola tabella base senza aggregazioni, raggruppamenti, ecc.).

2.11 Notazione alternativa per il Join (Orientata a SQL)

Le slide presentano anche una notazione alternativa per il join e la proiezione, più vicina a come SQL gestisce gli attributi qualificati e meno focalizzata sul natural join implicito.

- Si distinguono attributi con lo stesso nome usando la notazione `Relation.Attribute` (es. `EMPLOYEE.Wage`, `CHIEF.Wage`). Questo è standard in SQL.
- Le query join esplicite sono preferite (theta-join con $=$, o equi-join).
- Esempio con questa notazione (query per impiegati che guadagnano più del loro capo):

  $\pi_{\text{E.Number, E.Name, E.Wage, C.Number AS NumC, C.Name AS NameC, C.Wage AS WageC}}((\text{EMPLOYEE AS E} \bowtie_{\text{E.Dept = M.Dept}} ((\text{MANAGEMENT}$

  Questa notazione esplicita con alias (AS E, AS M, AS C) e condizioni di join è molto più vicina all'SQL reale:

```sql
SELECT E.Number, E.Name, E.Wage, C.Number AS NumC, C.Name AS NameC, C.Wa
FROM EMPLOYEE AS E
JOIN MANAGEMENT AS M ON E.Dept = M.Dept
JOIN EMPLOYEE AS C ON M.Chief = C.Number -- C alias for the chief emplo
WHERE E.Wage > C.Wage;
```

  L'uso degli alias e delle qualificazioni `E.Wage` vs `C.Wage` risolve l'ambiguità.

3 Calcolo Relazionale

Una famiglia di linguaggi **dichiarativi** basati sulla First Order Logic. Permettono di specificare le proprietà del risultato desiderato senza descrivere la procedura per ottenerlo.

3.1 Domain Relational Calculus (DRC)

- Basato su variabili che rangeano sui **domini dei valori** (es. una variabile `m` potrebbe assumere tutti i possibili valori del dominio dei numeri di impiegato).
- Sintassi generale: $\{x_1, x_2, \ldots, x_k | \text{Formula}(x_1, \ldots, x_k)\}$. Più precisamente, specificando gli attributi: $\{A_1 : x_1, \ldots, A_k : x_k | \text{Formula}(x_1, \ldots, x_k)\}$.
- Formula: una formula della logica del primo ordine che può includere:
  - Predicati che rappresentano le relazioni (es. $EMPLOYEE(m, n, a, w)$ significa "esiste una tupla nella relazione EMPLOYEE con valori m, n, a, w per i rispettivi attributi Number, Name, Age, Wage").
  - Predicati di confronto ($x > 40$, $x = y$).
  - Operatori logici ($\wedge, \vee, \neg$).
  - Quantificatori ($\forall, \exists$) sulle variabili (rangeano sull'intero dominio).
- Semantica: Il risultato è l'insieme delle tuple $(v_1, \ldots, v_k)$ di valori per le variabili $(x_1, \ldots, x_k)$ tali che la Formula è vera quando le variabili sono sostituite dai valori corrispondenti. Gli attributi della tupla risultante sono $A_1, \ldots, A_k$.
- Esempio: Restituire Numero, Nome, Età, Stipendio degli impiegati con stipendio ¿ 40.

$$\{\text{Number} : m, \text{Name} : n, \text{Age} : a, \text{Wage} : w \mid \text{EMPLOYEE}(m, n, a, w) \wedge w > 40\}$$

- Contro: Può essere prolisso. Permette di scrivere formule "domain-dependent" che non sono esprimibili in Algebra Relazionale (es. $A : x | R(A : x)$ - tutti i valori nel dominio di A che NON compaiono in R; l'algebra relazionale può solo operare sui valori *presenti* nelle relazioni). Si cerca di usare il DRC solo per query "domain-independent" (quelle esprimibili in RA).

3.2 Tuple Relational Calculus with Range Declarations (TRC-RD)

- Basato su variabili che rangeano su **intere tuple** di specifiche relazioni. Ogni variabile è dichiarata per rangeare su una relazione.
- Sintassi generale: {TargetList | RangeList | Formula}.
- RangeList: dichiara le variabili di tupla e su quale relazione rangeano (es. $e \in$ EMPLOYEE, o e(EMPLOYEE) nella notazione delle slide).
- TargetList: specifica quali attributi delle variabili di tupla compaiono nel risultato (es. $e.Number$, $e.Name$, o $e.(Number, Name, Age)$ o $e.*$ per tutti gli attributi).
- Formula: una formula logica che usa gli attributi delle variabili di tupla (es. $e.Wage > 40$).
- Semantica: L'insieme delle tuple formate dagli attributi specificati in TargetList, per le quali esiste un'assegnazione di tuple alle variabili in RangeList che soddisfa la Formula.
- Esempio: Restituire Numero, Nome, Età, Stipendio degli impiegati con stipendio ¿ 40.

$$\{e.* \mid e(\text{EMPLOYEE}) \mid e.\text{Wage} > 40\}$$

  Questo è più conciso del DRC per molte query.
- Esempio: Restituire numero, nome e età di tutti gli impiegati (proiezione).

$$\{e.(\text{Number, Name, Age}) \mid e(\text{EMPLOYEE}) \mid \text{TRUE}\}$$

  La formula TRUE indica che non ci sono ulteriori condizioni di selezione sulle tuple.

3.3 Equivalenza tra Calcolo e Algebra

- Il **Domain Relational Calculus** (limitato alle query domain-independent) e il **Relational Algebra** sono **equivalenti** nel loro potere espressivo. Qualsiasi query esprimibile nell'uno è esprimibile nell'altro.
- Il **Tuple Relational Calculus with Range Declarations** è anch'esso **equivalente** a DRC (domain-independent) e RA.
- Questo è un risultato fondamentale nella teoria dei database relazionali.

4 Oltre l'Algebra e il Calcolo Relazionale Standard

L'Algebra e il Calcolo Relazionale standard hanno dei limiti:

- Possono solo **estrarre** valori presenti nel database, non **computare nuovi valori** (es. calcolare un'imposta sul salario, fare una somma, una media). Le estensioni come l'aggregazione (SUM, AVG, COUNT) o le operazioni aritmetiche nelle proiezioni/selezioni sono state aggiunte in linguaggi pratici come SQL.
- Non possono esprimere query che richiedono un **numero arbitrario e a priori sconosciuto di join**. L'esempio classico è la **Transitive Closure**.

4.1 Transitive Closure

- Data una relazione binaria R su un insieme X (es. "c'è un volo diretto da X a Y"), la Transitive Closure R^+ è la più piccola relazione su X che contiene R ed è transitiva (se XRY e YRZ, allora XR^+Z).
- Nel caso dei voli, XR^+Y significa "è possibile volare da X a Y (con uno o più scali)".
- In Algebra Relazionale, per trovare tutti gli impiegati e i loro superiori diretti o indiretti (la relazione SUPERVISOR è la relazione base "direttamente supervisionato da"), dovrei fare:
  - Superiore diretto: SUPERVISOR
  - Superiore di 2 livelli: SUPERVISOR $\bowtie$ SUPERVISOR (rinominando gli attributi intermedi per il join)
  - Superiore di 3 livelli: SUPERVISOR $\bowtie$ SUPERVISOR $\bowtie$ SUPERVISOR
  - ...e fare l'unione di tutti questi risultati.
- Il problema è che non so a priori quanti livelli di supervisione (quanti join) potrebbero esistere in una data istanza del database. L'altezza della gerarchia potrebbe essere arbitraria.
- Pertanto, la Transitive Closure **non è esprimibile** in Algebra Relazionale o Calcolo Relazionale standard, che sono "first-order".
- SQL moderno ha estensioni per gestire questo, in particolare le **Common Table Expressions (CTE) ricorsive** (`WITH RECURSIVE`).

4.2 Datalog

- Un linguaggio di programmazione logica orientato ai database, basato su Prolog.
- Utilizza **regole** nella forma `head <- body` (congiunzione di predicati).
- Distingue tra:
  - **Predicati Estensionali:** Corrispondono alle tabelle base del database. Sono fatti noti a priori.
  - **Predicati Intensionali:** Definiti da regole. Corrispondono a viste derivate o a fatti che possono essere inferiti.
- Query: si chiede se un predicato è vero con un `?`. Esempio: `?richer(m, n, a, w).`
- Esempio di regola (equivalente a una Selection in RA):

```
richer(M, N, A, W) :- employee(M, N, A, W), W > 40.
-- Leggi: "Una tupla (M,N,A,W) e' nel predicato richer SE una tupla (M,
-- Corrisponde a: RICH PEOPLE = select_{Wage > 40}(EMPLOYEE)
```

- Esempio di regola (equivalente a un Join):

```
chief_of_rich(C) :- employee(M, N, A, W), W > 40, supervisor(C, M).
-- Leggi: "C e' un chief_of_rich SE (M,N,A,W) e' un employee, W > 40, E
-- Corrisponde a: pi_{Chief}(select_{Wage > 40}(EMPLOYEE naturaljoin SU
```

- Datalog supporta la **ricorsione** definendo predicati intensionali in termini di sé stessi. Questo permette di esprimere la Transitive Closure.

```
super_chief(E, C) :- supervisor(E, C). -- Regola base (supervisore dire
super_chief(E, C) :- supervisor(E, C'), super_chief(C', C). -- Regola r
-- super_chief(E, C) e' vero se C e' il supervisore diretto di E, OPPUR
```

- Questo insieme di regole definisce il predicato `super_chief(E, C)` come la Transitive Closure della relazione `supervisor(E, C)`.
- **Potere Espressivo di Datalog:**
  - Datalog non ricorsivo (con o senza negazione) è equivalente a RA/Calcolo standard.
  - Datalog **ricorsivo** (con negazione) è **più espressivo** di RA/Calcolo standard.

5 Conclusione

Abbiamo esplorato i fondamenti dei linguaggi di interrogazione relazionali: l'Algebra Relazionale (procedurale) e il Calcolo Relazionale (dichiarativo). Abbiamo visto che, per le query "first-order", questi linguaggi sono equivalenti. Abbiamo anche identificato i loro limiti, in particolare l'incapacità di esprimere computazioni (come aggregazioni) e la Transitive Closure. Linguaggi pratici come SQL hanno estensioni per superare queste limitazioni, mentre linguaggi teorici come Datalog mostrano come la ricorsione permetta di esprimere concetti come la Transitive Closure.