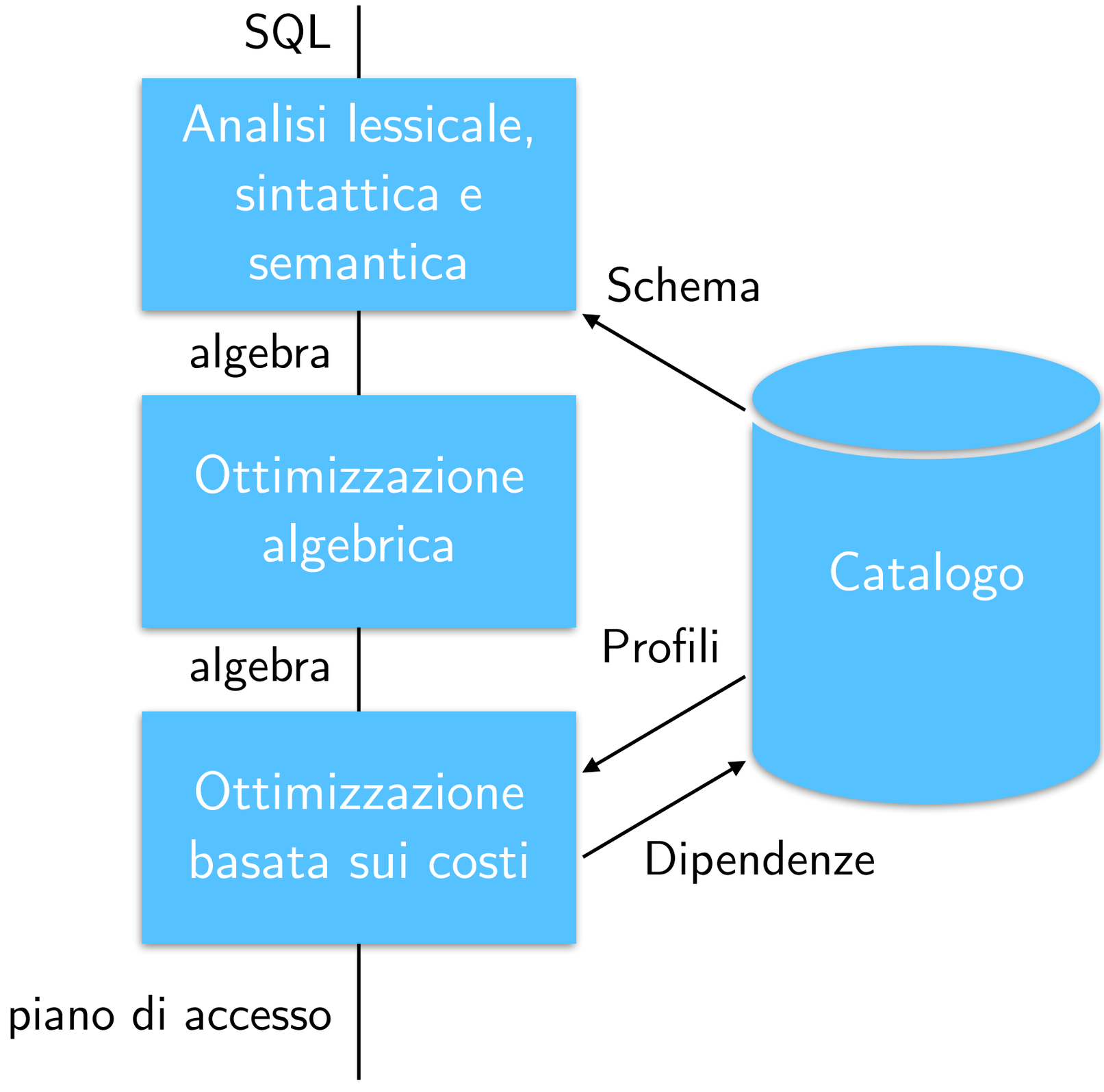
Ottimizzazione delle Interrogazioni

- ***Query processor*** (od **ottimizzatore**): un modulo del DBMS
- Più importante nei sistemi attuali che in quelli "vecchi" (gerarchici e reticolari):
  - Le interrogazioni sono espresse **ad alto livello** (ricordare il concetto di indipendenza dei dati):
    - insiemi di n-uple
    - poca proceduralità
- L'ottimizzatore sceglie la **strategia realizzativa** (di solito fra diverse alternative), a partire dall'istruzione SQL

Esecuzione delle Interrogazioni



Profili delle Relazioni

- **Informazioni quantitative**:
  - **cardinalità** di ciascuna relazione
  - **dimensioni** delle n-uple
  - **dimensioni** dei valori
    - **numero** di valori distinti degli attributi
    - valore **minimo** e **massimo** di ciascun attributo
- Sono **memorizzate** nel "catalogo" e **aggiornate** con comandi del tipo `update statistics`
- Utilizzate nella **fase finale** dell'ottimizzazione, per **stimare** le **dimensioni** dei **risultati intermedi**

Ottimizzazione Algebrica

- Il termine **ottimizzazione** è **improprio** (anche se efficace) perché il processo utilizza **euristiche**
- Si basa sulla nozione di **equivalenza**:
  - Due **espressioni** sono **equivalenti** se **producono lo stesso risultato** qualunque sia l'istanza attuale della base di dati
- I DBMS cercano di eseguire espressioni equivalenti a quelle date, ma **meno "costose"**
- Euristica fondamentale:
  - selezioni e proiezioni il più presto possibile (per **ridurre le dimensioni dei risultati intermedi**):
    - "**push selections down**"
    - "**push projections down**"

Esecuzione delle operazioni

- I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso
- Operatori fondamentali:
  - **accesso diretto**
  - **scansione**
- A livello più alto:
  - **ordinamento**
- Ancora più alto
  - **join**, l'operazione più costosa

Accesso diretto

- Può essere eseguito solo se le strutture fisiche lo permettono
  - **indici**
  - **strutture hash**

Accesso diretto basato su indice

- Efficace per interrogazioni (sulla “chiave dell’indice")
  - “**puntuali**” $(A_i = v)$
  - **su intervallo** $(v_1 \leq A_i \leq v_2)$
- Per **predicati congiuntivi**: si sceglie il più selettivo per l'accesso diretto e si verifica poi sugli altri dopo la lettura (e quindi in memoria centrale)
- Per **predicati disgiuntivi**: servono indici su tutti, ma conviene usarli se molto selettivi e facendo attenzione ai duplicati

Accesso diretto basato su hash

- Efficace per interrogazioni (sulla “chiave dell’indice")
  - “**puntuali**” $(A_i = v)$
  - **NON su intervallo** $(v_1 \leq A_i \leq v_2)$
- Per **predicati congiuntivi e disgiuntivi**, vale lo stesso discorso fatto per gli indici

Join

- L'operazione più costosa

- Vari metodi
  - I più noti:
    - **nested-loop**
    - **merge-scan**
    - **hash-based**

Nested Loop

- Per ogni n-upla nella tabella esterna si esaminano tutte le n-uple di quella interna per verificare la condizione di join
- Date R e S, si hanno due possibilità: R esterna o R interna
- Il costo, in termini di trasferimenti in memoria, dipende dal numero di accessi e dal fatto che la tabella interna sia piccola

Nested Loop

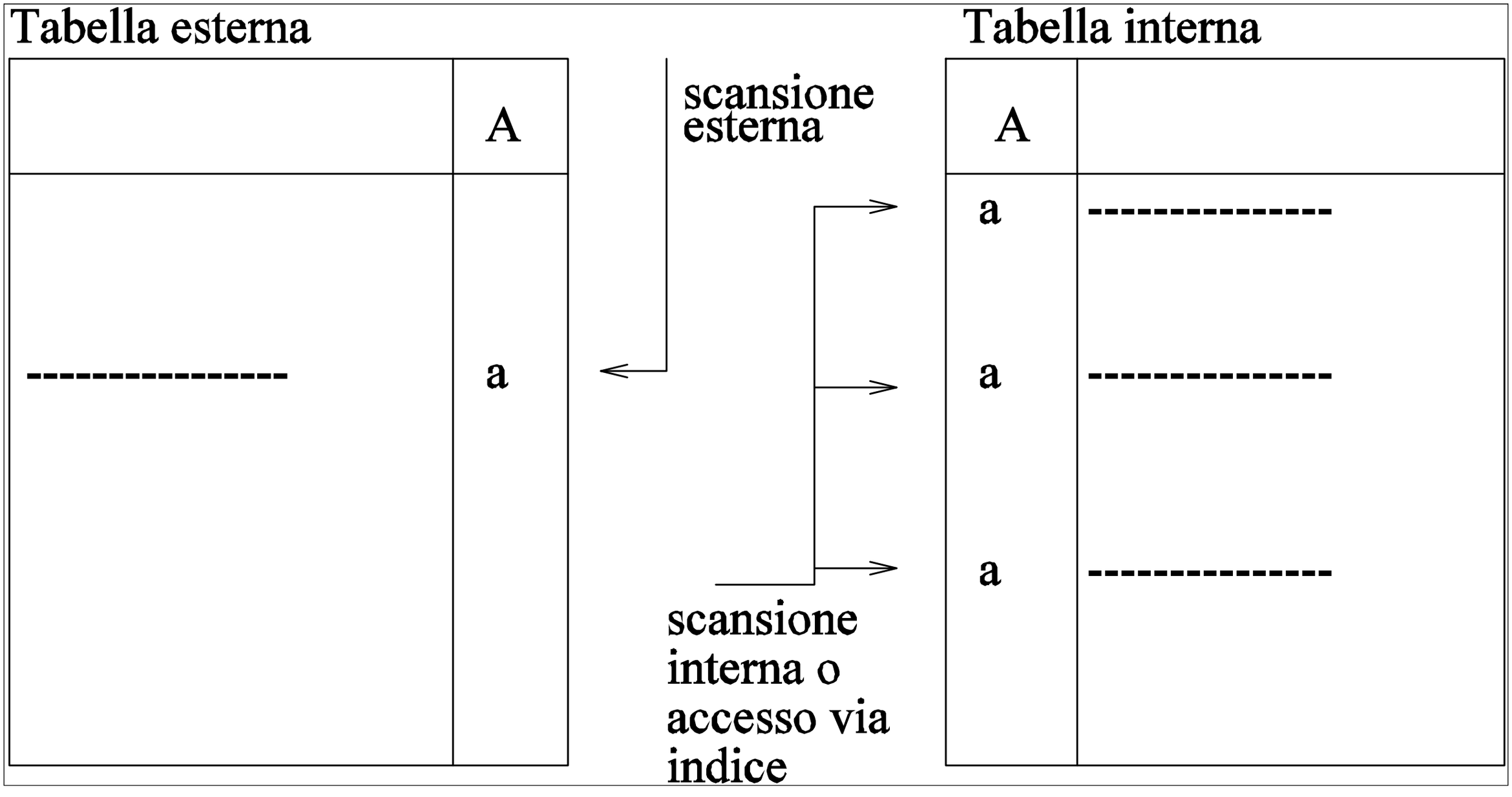


Tabella esterna
A
a
a
Tabella interna
A
a
a
a
scansione esterna
scansione interna o accesso via indice

Merge Scan

- Si ordinano le tabelle in base agli attributi di join
- Si trova l'elemento della seconda tabella da cui partire rispetto al primo elemento della prima tabella e poi si continua da lì
- Il costo è l'ordinamento

Merge Scan

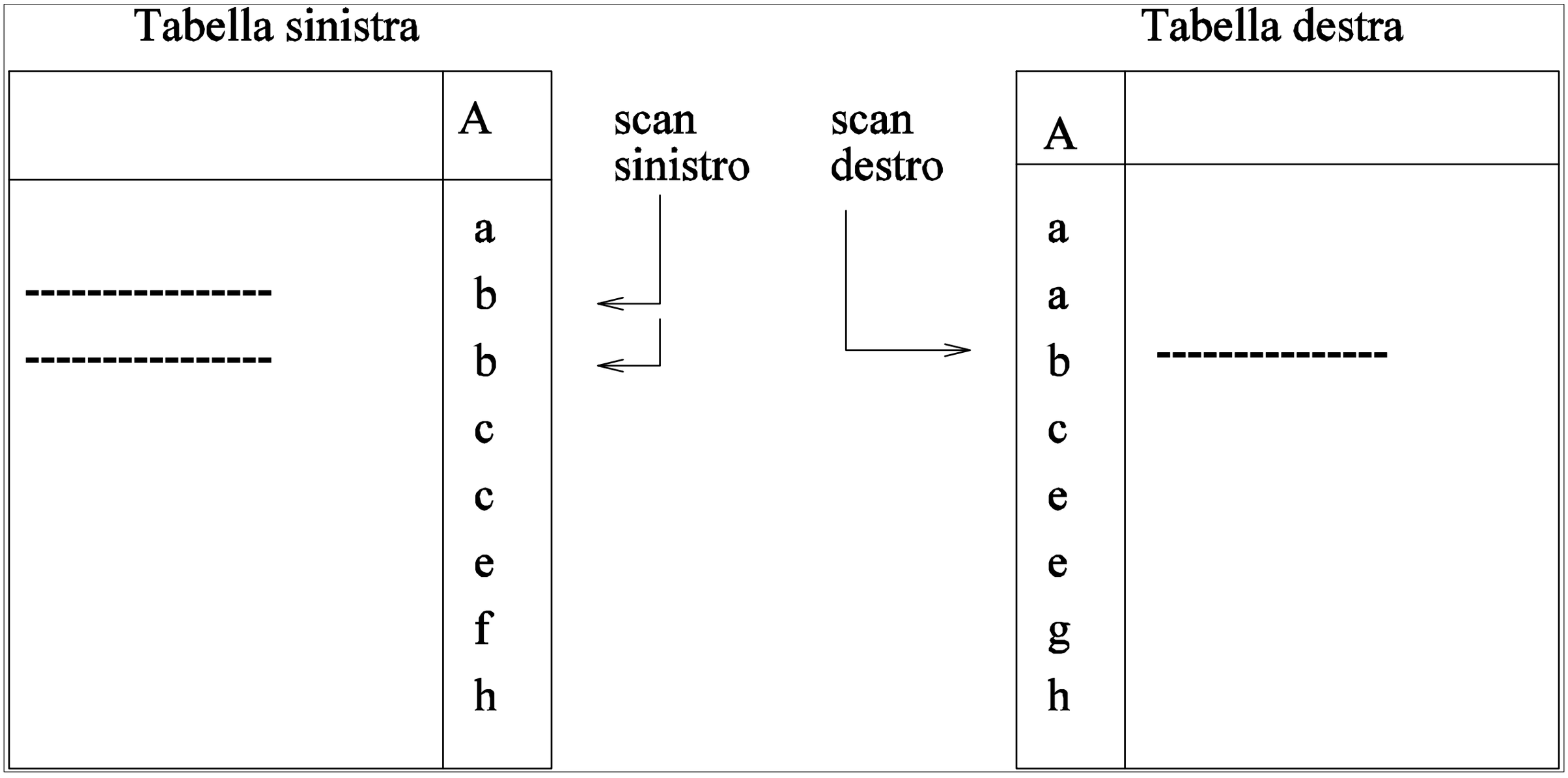
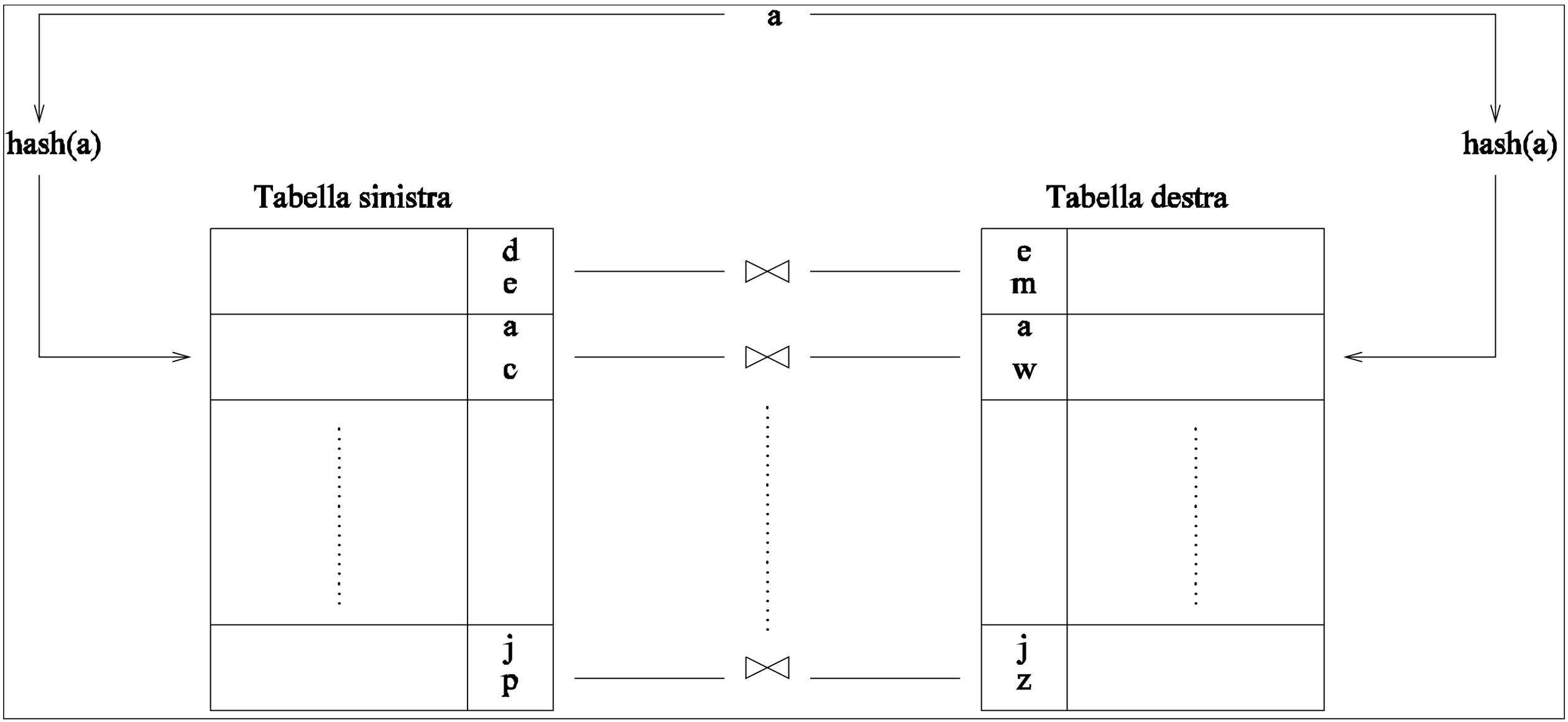


Tabella sinistra
A
a
b
b
c
c
e
f
h
scan sinistro
scan destro
Tabella destra
A
a
a
b
c
e
e
g
h

Hash Join

- Utile solo per **join naturale** e **equi-join**
- La funzione hash h sull'attributo di join è usata per **partizionare le n-uple** di entrambe le relazioni
- Le partizioni sono inserite in tabelle aggiuntive, h produce partizioni di S tali che ognuna sta in memoria, R è partizionata di conseguenza
- Serve memoria ed è migliore del nested loop nel caso di equi-join

Hash Join


a
hash(a)
hash(a)
Tabella sinistra
Tabella destra
d
e
a
c
j
p
e
m
a
w
j
z

Ottimizzazione basata sui costi

- Un problema articolato, con scelte relative a:
  - **operazioni da eseguire** (es.: scansione o accesso diretto?)
  - **ordine delle operazioni** (es. join di tre relazioni; ordine?)
  - **dettagli del metodo** (es.: quale metodo di join)

- Architetture **parallele** e **distribuite** aprono ulteriori **gradi di libertà**

Misura del costo di una query

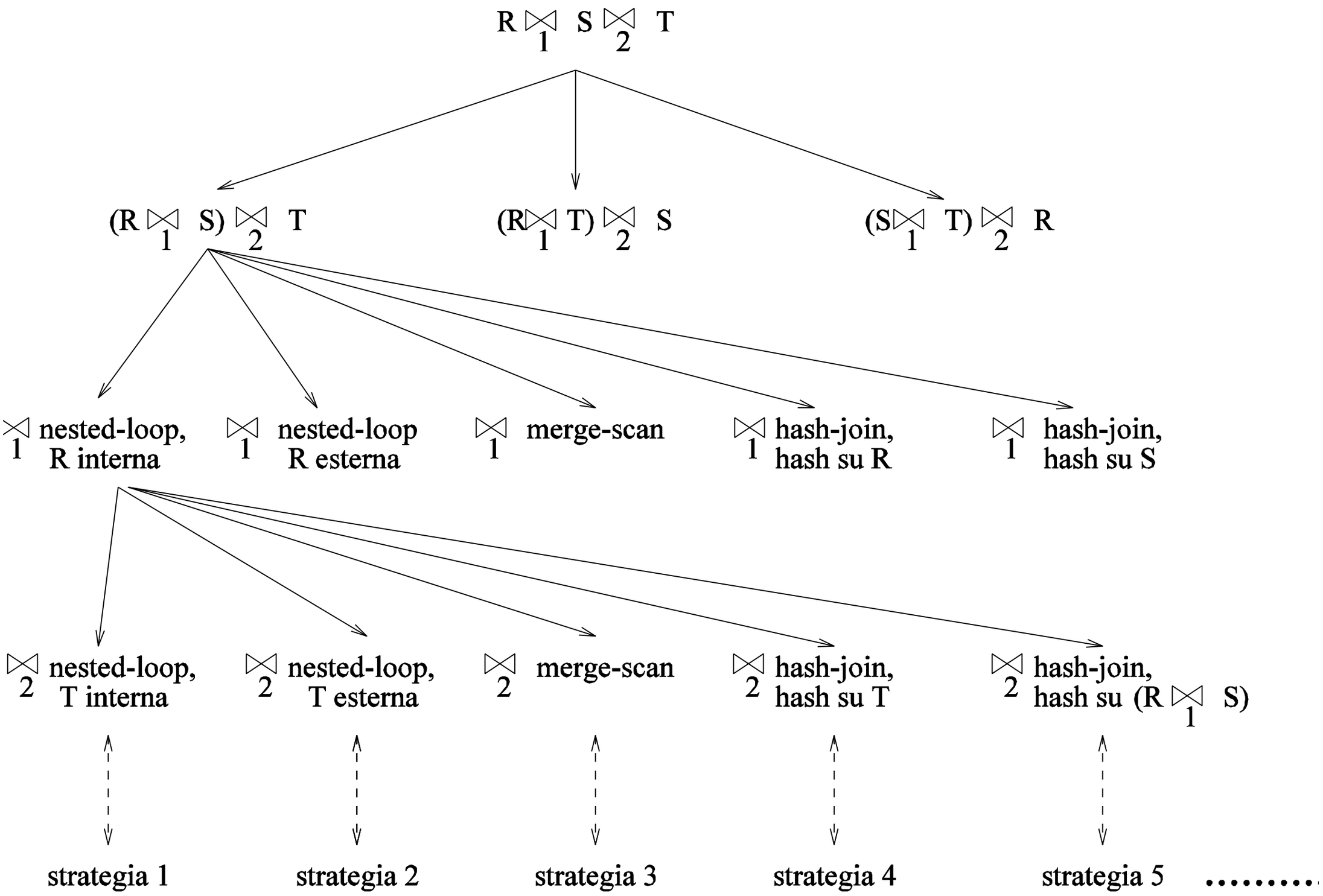
- Molti fattori contribuiscono al costo di una query, cioè il tempo necessario per avere la risposta:
  - Gli accessi al disco, il tempo di CPU o il tempo di rete
  - L'accesso al disco è il tempo predominante ed è anche facilmente calcolabile considerando
    - Numero di scansioni
    - Numero di letture
    - Numero di scritture
- Oltre al numero di trasferimenti bisogna considerare anche la memoria che serve per memorizzare i risultati intermedi
  - Ad esempio l'hash join necessita di tabelle intermedie
- Operazioni a più operandi devono essere eseguite un passo alla volta

Processo di ottimizzazione

- Si costruisce un albero di decisione con le varie alternative ("piani di esecuzione")
- Si valuta il costo di ciascun piano
- Si sceglie il piano di costo minore

- L'ottimizzatore trova di solito una "buona" soluzione, non necessariamente l'"ottimo"

Esempio


R ⋈1 S ⋈2 T
(R ⋈1 S) ⋈2 T
(R ⋈1 T) ⋈2 S
(S ⋈1 T) ⋈2 R
⋈1 nested-loop, R interna
⋈1 nested-loop R esterna
⋈1 merge-scan
⋈1 hash-join, hash su R
⋈1 hash-join, hash su S
⋈2 nested-loop, T interna
⋈2 nested-loop, T esterna
⋈2 merge-scan
⋈2 hash-join, hash su T
⋈2 hash-join, hash su (R ⋈1 S)
strategia 1
strategia 2
strategia 3
strategia 4
strategia 5
.....