

# Query processing and optimization

# Definitions

- Query processing
  - translation of query into low-level activities
  - evaluation of query
  - data extraction
- Query optimization
  - selecting the most efficient query evaluation

# Query Processing (1/2)

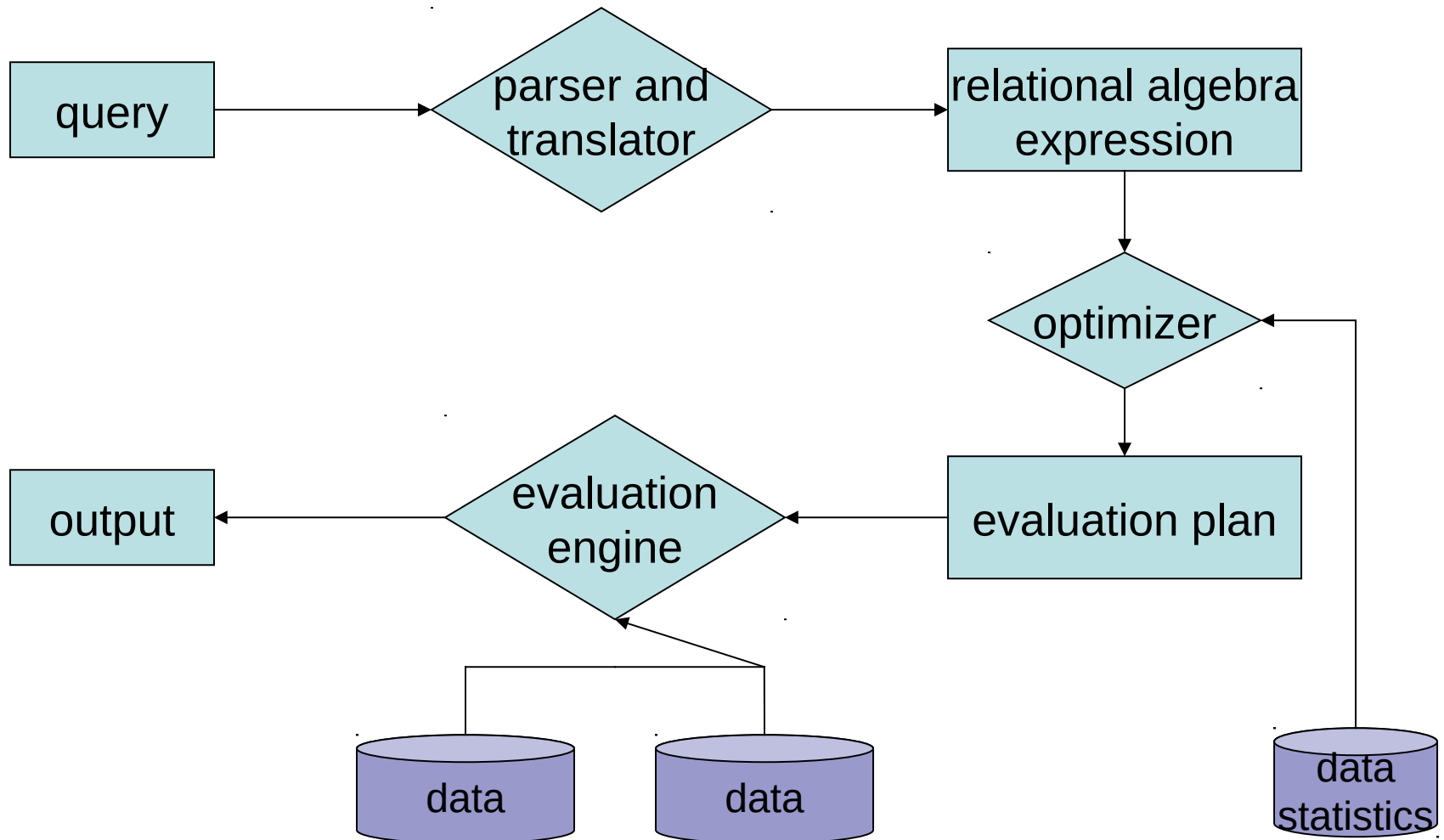
- SELECT \* FROM student WHERE name=Paul
- Parse query and translate
  - check syntax, verify names, etc
  - translate into relational algebra (RDBMS)
  - create **evaluation plans**
- Find best plan (optimization)
- Execute plan

student	
<u>cid</u>	name
00112233	Paul
00112238	Rob
00112235	Matt

takes	
<u>cid</u>	<u>courseid</u>
00112233	312
00112233	395
00112235	312

course	
<u>courseid</u>	<u>coursename</u>
312	Advanced DBs
395	Machine Learning

# Query Processing (2/2)



# Relational Algebra (1/2)

- Query language
- Operations:
  - select:  $\sigma$
  - project:  $\pi$
  - union:  $\cup$
  - difference:  $-$
  - product:  $\times$
  - join:  $\bowtie$
- Extended relational algebra operations:

# Relational Algebra (2/2)

- `SELECT * FROM student WHERE name=Paul`  
 –  $\sigma_{\text{name=Paul}}(\text{student})$
- $\pi_{\text{name}}(\sigma_{\text{cid}<00112235}(\text{student}))$
- $\pi_{\text{name}}(\sigma_{\text{coursename=Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$

student	
<u>cid</u>	name
00112233	Paul
00112238	Rob
00112235	Matt

takes	
<u>cid</u>	<u>courseid</u>
00112233	312
00112233	395
00112235	312

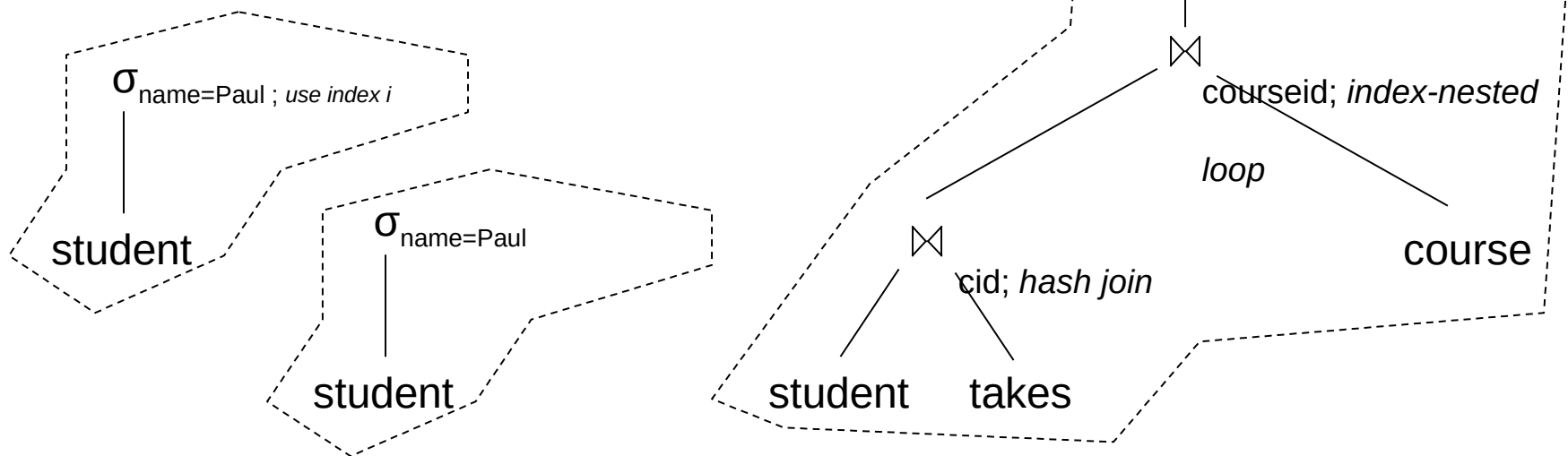
course	
<u>courseid</u>	<u>coursename</u>
312	Advanced DBs
395	Machine Learning

# Why Optimize?

- Many alternative options to evaluate a query
  - $\pi_{\text{name}}(\sigma_{\text{course name}=\text{Advanced DBs}}((\text{student}_{\text{cid}} \bowtie \text{takes}) \bowtie \text{course}_{\text{courseid}}))$
  - $\pi_{\text{name}}((\text{student}_{\text{cid}} \bowtie \text{takes}) \bowtie \text{course}_{\text{courseid}} \sigma_{\text{course name}=\text{Advanced DBs}}))$
- Several options to evaluate a single operation
  - $\sigma_{\text{name}=\text{Paul}}(\text{student})$ 
    - scan file
    - use secondary index on student.name
- Multiple access paths
  - access path: how can records be accessed (using index; which index?; etc.)

# Evaluation plans

- Specify which access path to follow
- Specify which algorithm to use to evaluate operator
- Specify how operators interleave
- Optimization:
  - estimate the cost of each plan (not all plans)
  - select plan with lowest estimated cost





# Estimating Cost

- What needs to be considered:
  - Disk I/Os
    - sequential (reading neighbouring pages is faster)
    - random
  - CPU time
  - Network communication
- What are we going to consider:
  - Disk I/Os
    - page (data block) reads/writes
  - Ignoring cost of writing final output

# Operations and Costs (1/2)

- Operations:  $\sigma$ ,  $\pi$ ,  $\cup$ ,  $\cap$ ,  $-$ ,  $\times$ ,  $\bowtie$
- Costs:
  - $N_R$ : number of records in R (other notation:  $T_R$  or  $T(R)$  -> tuple)
  - $L_R$ : size of record in R (length of record)
  - $bf_R$ : blocking factor (other notation:  $F_R$ )
    - number of records in a page (datablock)
  - $B_R$ : number of pages to store relation R
  - $V(R, A)$ : number of distinct values of attribute A in R  
other notation:  $I_A(R)$  (Image size)
  - $SC(R, A)$ : selection cardinality of A in R (number of matching rec.)
    - A key:  $SC(R, A)=1$
    - A nonkey:  $SC(R, A)= T_R / V(R, A)$  (uniform distribution assumption)
  - $HT_i$ : number of levels in index I (-> height of tree)
  - rounding up fractions and logarithms

# Operations and Costs (2/2)

- relation *takes*
  - 7000 tuples
  - student cid 8 bytes
  - course id 4 bytes
  - 40 courses
  - 1000 students
  - page size 512 bytes
  - output size (in pages) of query:  
**which students take the Advanced DBs course?**
    - $T_{\text{takes}} = 7000$
    - $V(\text{courseid}, \text{takes}) = 40$
    - $SC(\text{courseid}, \text{takes}) = \text{ceil}(T_{\text{takes}}/V(\text{courseid}, \text{takes})) = \text{ceil}(7000/40) = 175$
    - $bf_{\text{takes}} = \text{floor}(512/12) = 42$     $B_{\text{takes}} = 7000/42 = 167$  pages
    - $bf_{\text{output}} = \text{floor}(512/8) = 64$     $B_{\text{output}} = 175/64 = 3$  pages

# Cost of Selection $\sigma$ (1/2)

- Linear search
  - read all pages, find records that match (assuming equality search)
  - average cost:
    - nonkey (multiple occurrences):  $B_R$ , key:  $0.5 \cdot B_R$
- Binary search
  - on ordered field
  - average cost:  $\lceil \log_2 B_R \rceil + m$ 
    - $m$  additional pages to be read (first found then read the duplicates)
    - $m = \text{ceil}( SC(A,R)/bf_R ) - 1$
- Primary/Clustered Index (B+ tree)
  - average cost:
    - single record:  $HT_i + 1$
    - multiple records:  $HT_i + \text{ceil}( SC(R,A)/bf_R )$

# Cost of Selection $\sigma$ (2/2)

- Secondary Index (B+ tree)
  - average cost:
    - key field:  $HT_i + 1$
    - nonkey field
      - worst case  $HT_i + SC(A,R)$
      - linear search more desirable if many matching records !!!

# Complex selection $\sigma_{\text{expr}}$

- conjunctive selections:  $\sigma_{\theta_1 \wedge \theta_2 \dots \wedge \theta_n}$ 
  - perform simple selection using  $\theta_i$  with the lowest evaluation cost
    - e.g. using an index corresponding to  $\theta_i$
    - apply remaining conditions  $\theta$  on the resulting records
    - $\sigma_{cid > 00112233 \wedge courseid = 312}(\text{takes})$
    - cost: the cost of the simple selection on selected  $\theta$
  - multiple indices
    - select indices that correspond to  $\theta_i$ s
    - scan indices and return RIDs (ROWID in Oracle)
    - answer: intersection of RIDs
    - cost: the sum of costs + record retrieval
- disjunctive selections:  $\sigma_{\theta_1 \vee \theta_2 \dots \vee \theta_n}$ 
  - multiple indices
    - union of RIDs
  - linear search

# Projection and set operations

- SELECT DISTINCT cid FROM takes
  - $\pi$  requires duplicate elimination
  - sorting
- set operations require duplicate elimination
  - $R \cap S$
  - $R \cup S$
  - sorting

# Sorting

- efficient evaluation for many operations
- required by query:
  - `SELECT cid, name FROM student ORDER BY name`
- implementations
  - **internal sorting** (if records fit in memory)
  - **external sorting**  
(that's why we need temporary space on disk)



# External Sort-Merge Algorithm (1/3)

- Sort stage: create sorted *runs*

i=0;

repeat

    read  $M$  pages of relation  $R$  into memory ( **$M$ : size of Memory**)

    sort the  $M$  pages

    write them into file  $R_i$

    increment  $i$

until no more pages

$N = i$                // number of runs

# External Sort-Merge Algorithm (2/3)

- Merge stage: merge sorted *runs*

//assuming  $N < M$  ( $N \leq M-1$  we need 1 output buffer)

allocate a page for each run file  $R_i$       //  $N$  pages allocated

read a page  $P_i$  of each  $R_i$

repeat

    choose first record (in sort order) among  $N$  pages, say from page  $P_j$

    write record to output and delete from page  $P_j$

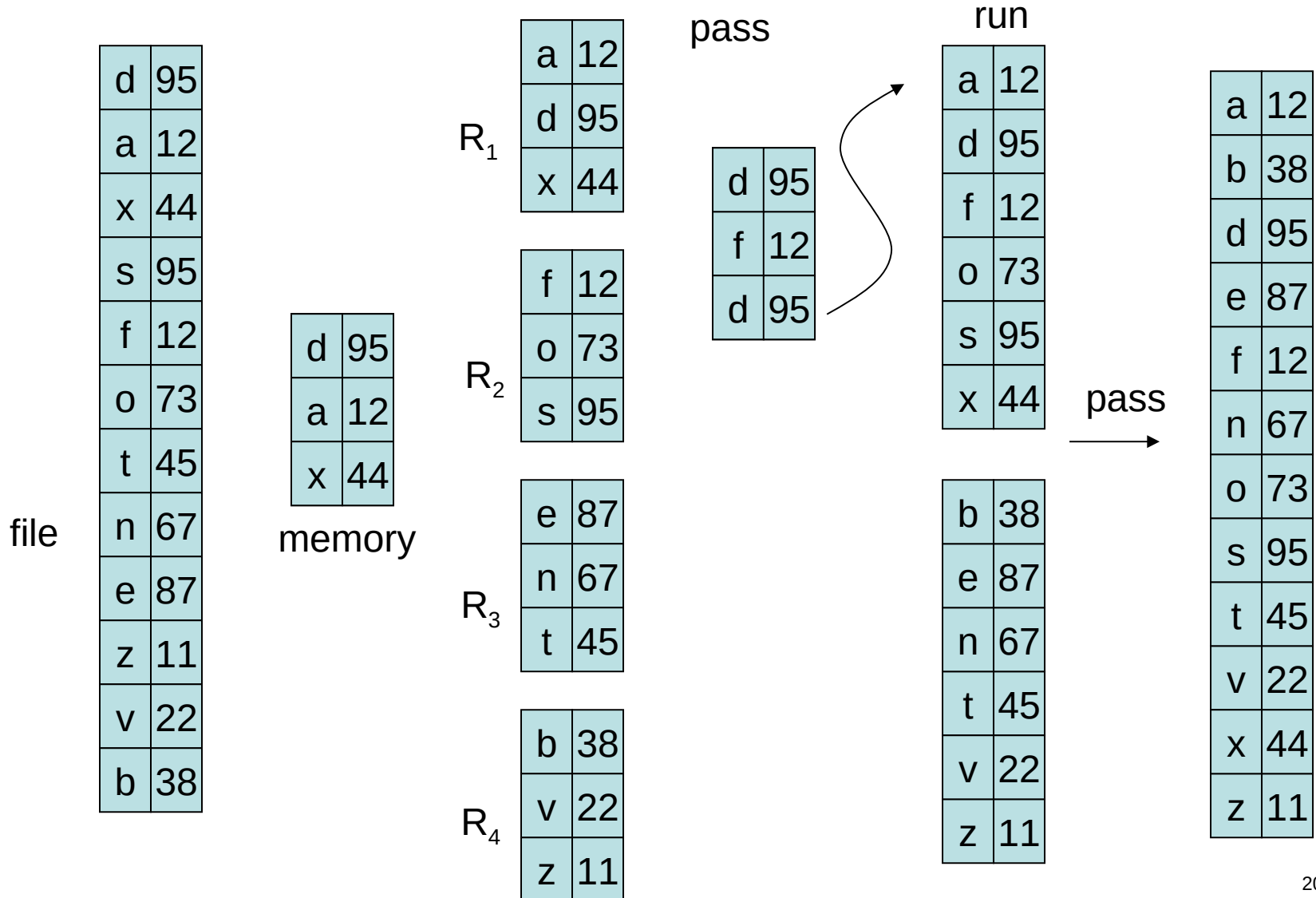
    if page is empty read next page  $P_j'$  from  $R_j$

until all pages are empty

# External Sort-Merge Algorithm (3/3)

- Merge stage: merge sorted *runs*
- What if  $N \geq M$  ?
  - perform multiple *passes*
  - each *pass* merges  $M-1$  runs until relation is processed
  - in next pass number of runs is reduced
  - final *pass* generated sorted output

# Sort-Merge Example



# Sort-Merge cost

- $B_R$  the number of pages of R
- Sort stage:  $2 * B_R$ 
  - read/write relation
- Merge stage:
  - initially  $\left\lceil \frac{B_R}{M} \right\rceil$  runs to be merged
  - each pass  $M-1$  runs sorted
  - thus, total number of passes:  $\left\lceil \log_{M-1} \left( \frac{B_R}{M} \right) \right\rceil$
  - at each pass  $2 * B_R$  pages are read/written
    - read/write relation ( $B_R + B_R$ )
    - apart from final write ( $B_R$ )
- Total cost:
  - $2 * B_R + 2 * B_R * \left\lceil \log_{M-1} \left( \frac{B_R}{M} \right) \right\rceil - B_R$       eg.  $B_R = 1000000, M=100$

# Projection

- $\pi_{A_1, A_2 \dots} (R)$
- remove unwanted attributes
  - scan and drop attributes
- remove duplicate records
  - sort resulting records using all attributes as sort order
  - scan sorted result, eliminate duplicates (adjacent)
- cost
  - initial scan + **sorting** + final scan

# Join

- $\pi_{\text{name}}(\sigma_{\text{course name}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$
- implementations
  - nested loop join
  - block-nested loop join
  - indexed nested loop join
  - sort-merge join
  - hash join

# Nested loop join (1/2)

- $R \bowtie S$

for each tuple  $t_R$  of  $R$

    for each  $t_S$  of  $S$

        if ( $t_R$   $t_S$  match) output  $t_R.t_S$

    end

end

- Works for any join condition
- $S$  inner relation
- $R$  outer relation



# Nested loop join (2/2)

- Costs:
  - **best case** when smaller relation fits in memory
    - use it as inner relation
    - $B_R + B_S$
  - **worst case** when memory holds one page of each relation
    - S scanned for each tuple in R
    - $T_R * B_S + B_R$

# Block nested loop join (1/2)

```
for each page  $X_R$  of R
  for each page  $X_S$  of S
    for each tuple  $t_R$  in  $X_R$ 
      for each  $t_S$  in  $X_S$ 
        if ( $t_R$   $t_S$  match) output  $t_R.t_S$ 
      end
    end
  end
end
end
```

# Block nested loop join (2/2)

- Costs:
  - **best case** when smaller relation fits in memory
    - use it as inner relation
    - $B_R + B_S$
  - **worst case** when memory holds one page of each relation
    - S scanned for each page in R
    - $B_R * B_S + B_R$

# Block nested loop join (an improvement)

Memory size:  $M$

for each  $M - 1$  page size chunk  $M_R$  of  $R$

for each page  $X_S$  of  $S$

for each tuple  $t_R$  in  $M_R$

for each  $t_S$  in  $X_S$

if ( $t_R$   $t_S$  match) output  $t_R.t_S$

end

end

end

end

# Block nested loop join (an improvement)

- Costs:
  - best case when smaller relation fits in memory
    - use it as inner relation
    - $B_R + B_S$
  - general case
    - S scanned for each M-1 size chunk in R
    - $(B_R / (M-1)) * B_S + B_R$

# Indexed nested loop join

- $R \bowtie S$
- Index on inner relation (S)
- for each tuple in outer relation (R) *probe* index of inner relation
- Costs:
  - $B_R + T_R * c$ 
    - $c$  the cost of index-based selection of inner relation
  - $c \approx T(s)/V(s,A)$   
(if A is the join column and index is kept in memory)
  - relation with fewer records as outer relation

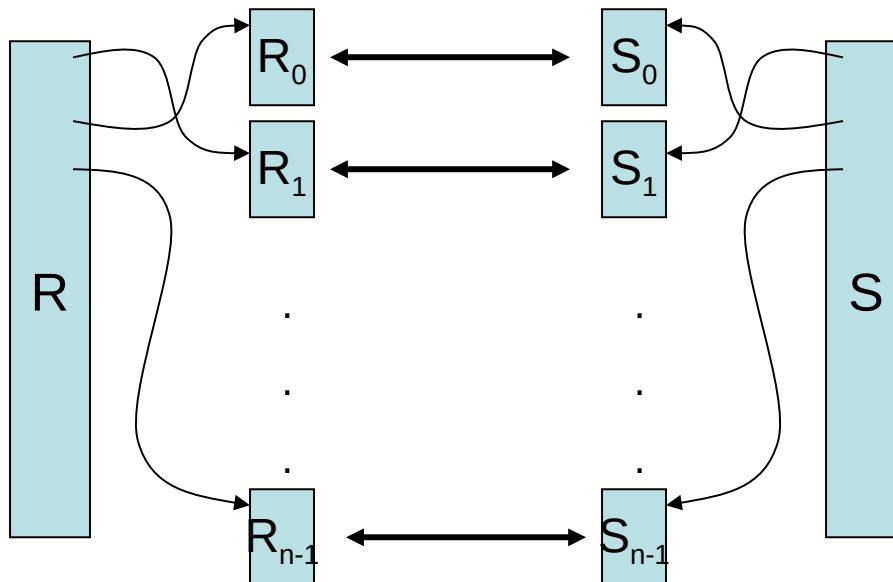
# Sort-merge join

- $R \bowtie S$
- Relations sorted on the join attribute
- Merge sorted relations
  - pointers to first record in each relation
  - read in a group of records of S with the same values in the join attribute
  - read records of R and process
- Relations in sorted order to be read once
- Cost:
  - cost of sorting +  $B_S + B_R$

→	d	D	→	e	67
	e	E		e	87
	x	X		n	11
	v	V		v	22
				z	38

# Hash join

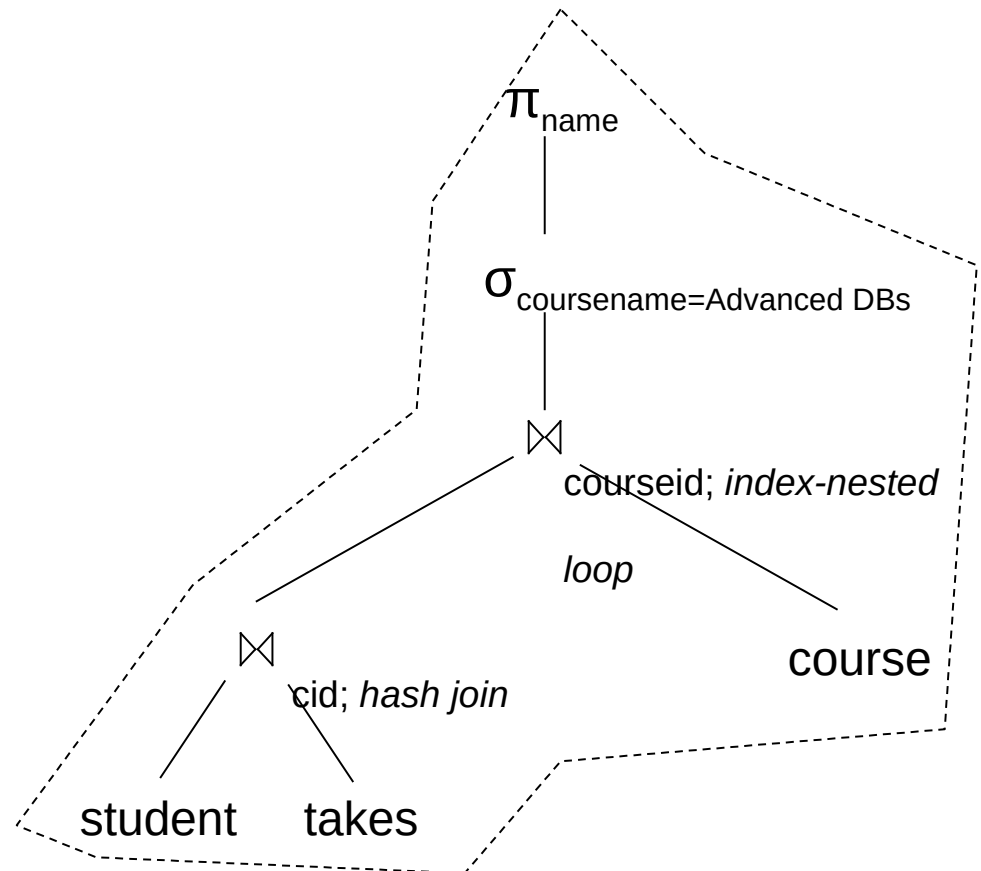
- $R \bowtie S$
- use  $h_1$  on joining attribute to map records to partitions that fit in memory
  - records of  $R$  are partitioned into  $R_0 \dots R_{n-1}$
  - records of  $S$  are partitioned into  $S_0 \dots S_{n-1}$
- join records in corresponding partitions
  - using a hash-based indexed block nested loop join
- **Cost:  $2 \cdot (B_R + B_S) + (B_R + B_S)$**





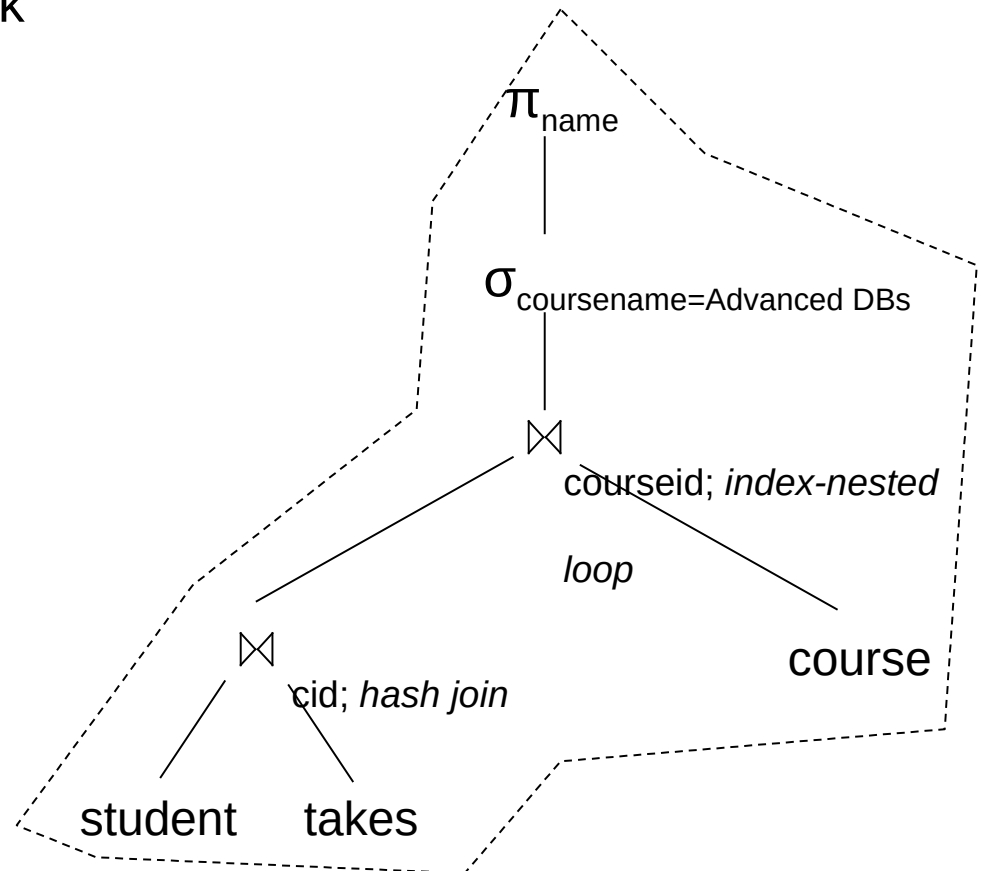
# Evaluation

- evaluate multiple operations in a plan
- materialization
- pipelining



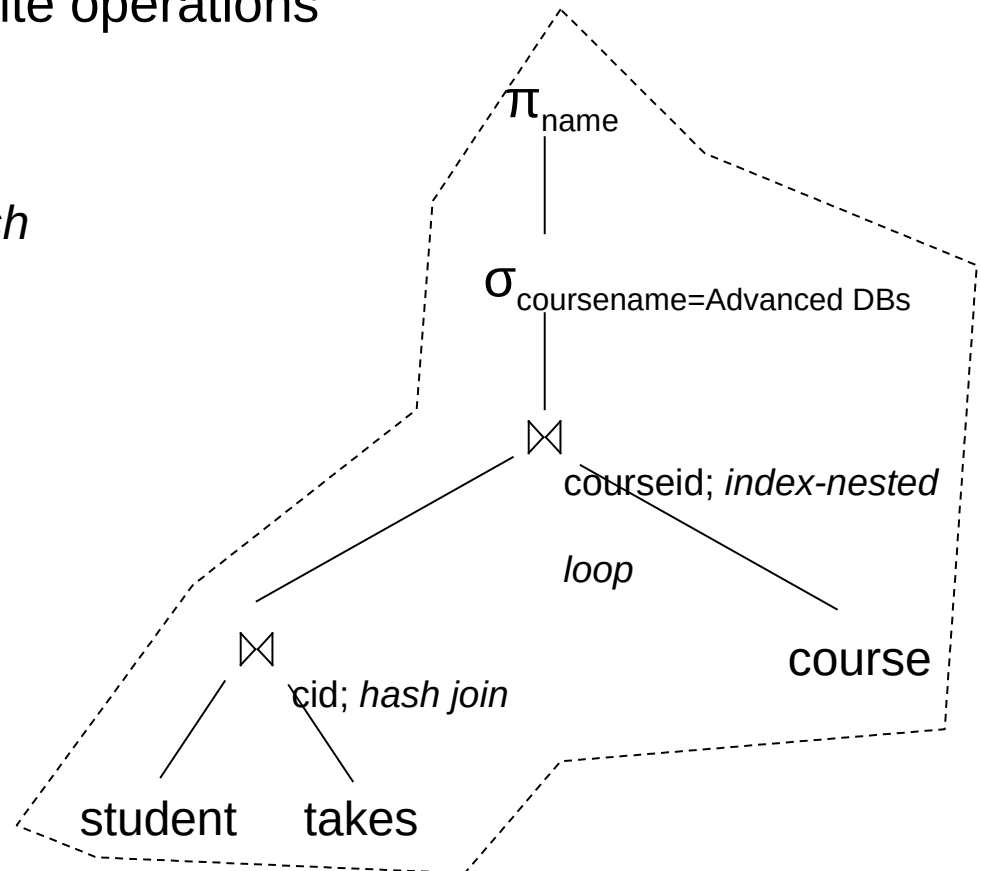
# Materialization

- create and read temporary relations
- create implies writing to disk
  - more page writes



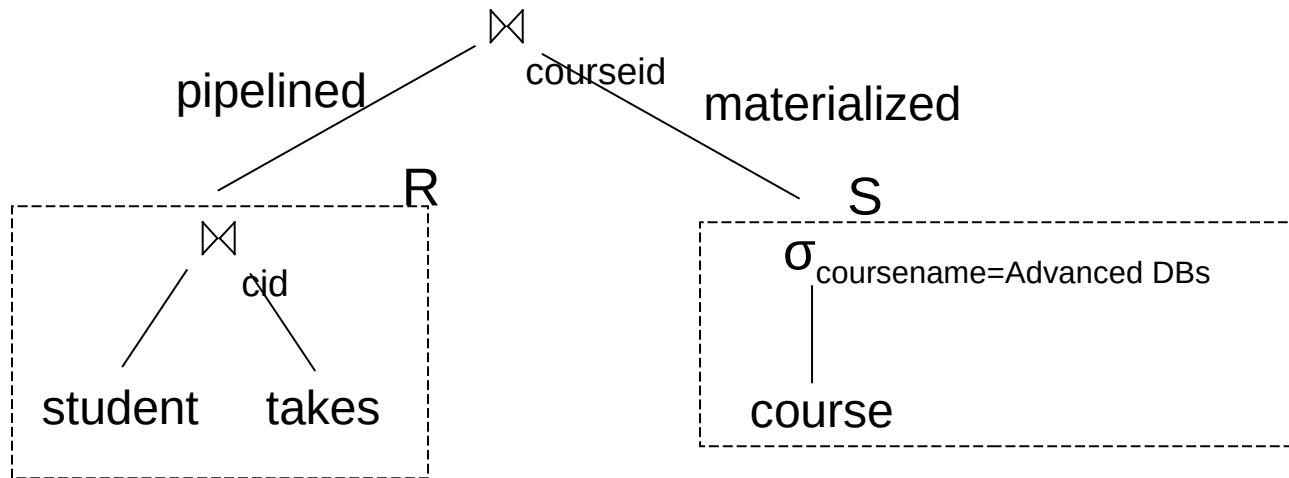
# Pipelining (1/2)

- creating a pipeline of operations
- reduces number of read-write operations
- implementations
  - demand-driven - data *pull*
  - producer-driven - data *push*



# Pipelining (2/2)

- can pipelining always be used?
- any algorithm?
- cost of  $R \bowtie S$ 
  - materialization and hash join:  $B_R + 3(B_R + B_S)$
  - pipelining and indexed nested loop join:  $T_R * HT_i$

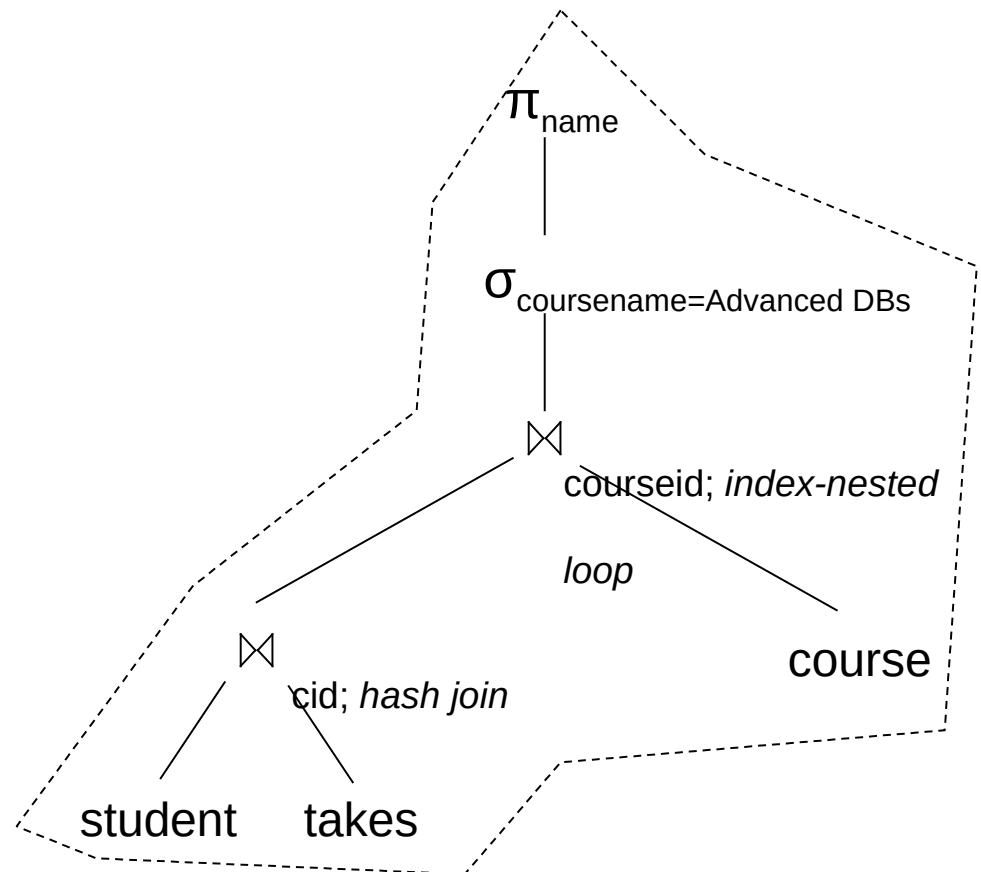


# Choosing evaluation plans

- cost based optimization
- enumeration of plans
  - $R \bowtie S \bowtie T$ , 12 possible orders ( $3! * 2$ )  
 $(R \bowtie S) \bowtie T$ ,  $R \bowtie (S \bowtie T)$
- cost estimation of each plan
- overall cost
  - cannot optimize operation independently

# Cost estimation

- operation ( $\sigma$ ,  $\pi$ ,  $\bowtie$  ...)
- implementation
- size of inputs
- size of outputs
- sorting

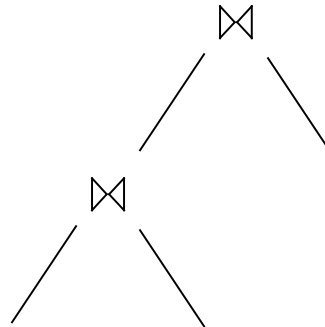


# Expression Equivalence

- conjunctive selection decomposition
  - $\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$
- commutativity of selection
  - $\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$
- combining selection with join and product
  - $\sigma_{\theta_1}(R \bowtie S) = \dots \quad R \bowtie_{\theta_1} S = \dots$
- commutativity of joins
  - $R \bowtie_{\theta_1} S = S \bowtie_{\theta_1} R$
- distribution of selection over join
  - $\sigma_{\theta_1 \wedge \theta_2}(R \bowtie S) = \sigma_{\theta_1}(R) \bowtie \sigma_{\theta_2}(S)$
- distribution of projection over join
  - $\pi_{A_1, A_2}(R \bowtie S) = \pi_{A_1}(R) \bowtie \pi_{A_2}(S)$
- associativity of joins:  $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

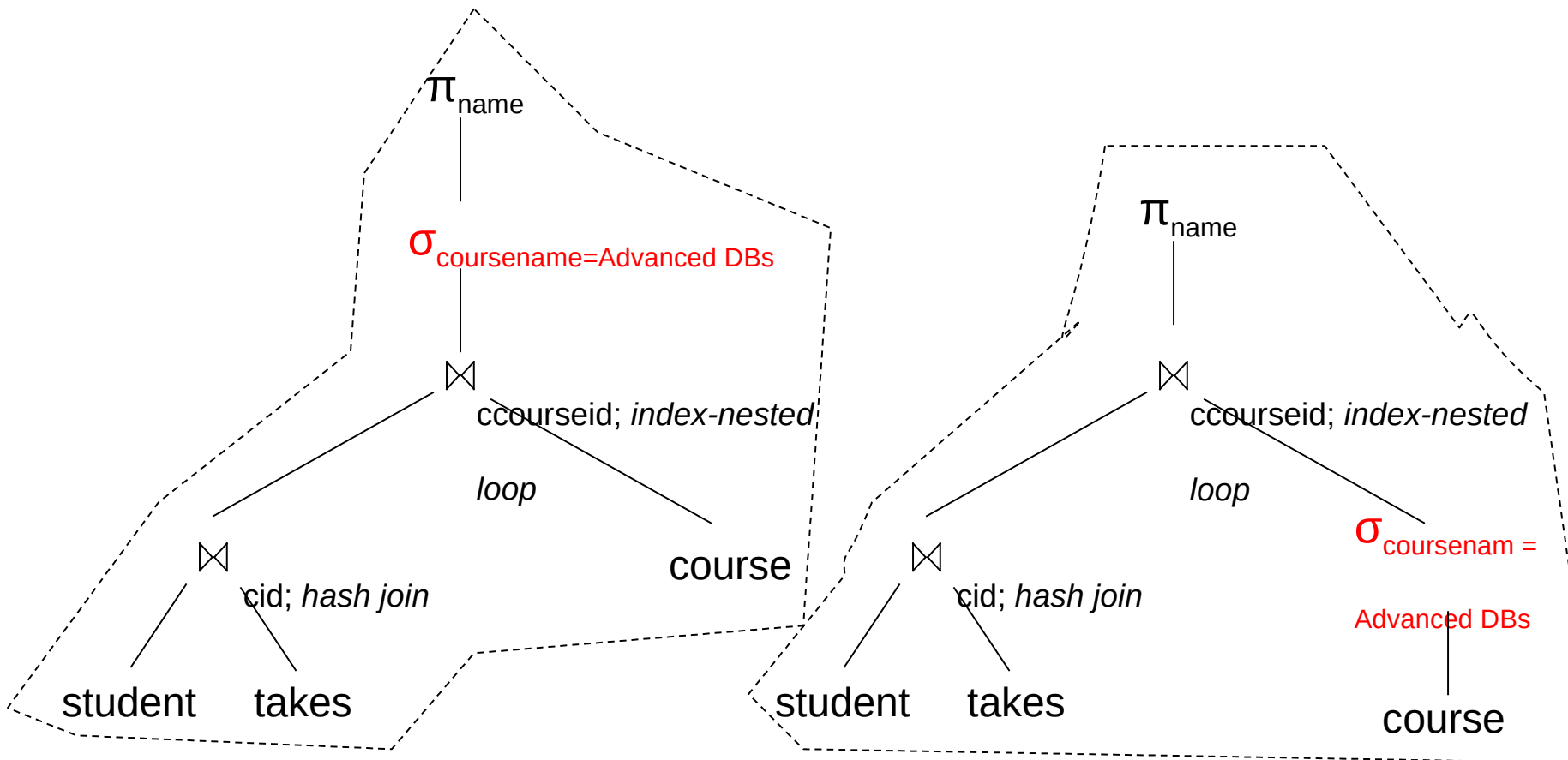
# Cost Optimizer (1/2)

- transforms expressions
  - equivalent expressions
  - heuristics, *rules of thumb*
    - perform **selections early**
    - perform **projections early**
    - replace products followed by selection  $\sigma (R \times S)$  with joins  $R \bowtie S$
    - start with joins, selections with smallest result
      - create **left-deep join trees**





# Cost Optimizer (2/2)



# Summary

- Estimating the cost of a single operation
- Estimating the cost of a query plan
- Optimization
  - choose the most efficient plan