Relational Operators

COURSE 6: Databases

Relational model

Relational model

- Codd rules 1985 → Is DBMS relational? If yes, to what degree?
- [2]

Relational Integrity constraints

RELATIONS

OPERATORS

Relational model

- Database = collection of RELATIONS
  - relation in relational model ≠ relationship in ERD.
  - relation in relation model < -- > table with lines and columns

- Relation Schema: A relation schema represents the name of the relation with its attributes.

- Attribute domain – Each attribute has some pre-defined values.

Relational Integrity constraints

RELATIONS

OPERATORS

- Relational schema $R(A_1, A_2, \ldots, A_n)$
- $R \subset D_1 \times D_2 \times \cdots \times D_n, D_i \ domain$

- Example
  - Participant(participant_id, last_name, first_name)
  - A1 - - participant_id D1 - - integer size 6
  - A2 - - last_name D2 - - string, length 20
  - A3 - - first_name D3 - - string, length 20

Relational Integrity constraints

RELATIONS

OPERATORS

- Domain constraints
  - “the value of each attribute must be unique”, specifies data types: integers, real numbers, characters, Booleans; variable length for strings, numbers etc.
- Key constraint
  - Unique + not null -- PK
- Referential integrity constraints
  - the value of a FK is null or it corresponds to the value of a PK.

Relational Integrity constraints

RELATIONS

OPERATORS

- UNION, INTERSECT, PRODUCT, DIFFERENCE

- PROJECT
- SELECT
- JOIN
- DIVISION

Relational operators

Relational algebra

Relational algebra

- Operands → relations (tables)

- Operators → operate on a relation/combine two relations and obtain as a result a new relation
  → compositional

PROJECT, SELECT, DIFFERENCE, PRODUCT, UNION

Derived operators: JOIN, DIVISION, INTERSECT

Project

- Unary operator
- Notations: PROJECT(R, X) or $\Pi_X\ (R)$
  - R is a relation; X is a set of attributes of R
- The result is a relation with a subset of the attributes X.
- Eliminating attributes from R may lead to duplicate rows. Hence after eliminating attributes, project also eliminates duplicate lines.

PROJECT(R, X) $\Pi_X\,(R)$

R

A	B	C
x1	y1	z1
x1	y1	z2
x1	y2	z2
x2	y2	z1

$\longrightarrow$

A	C
x1	z1
x1	z2
x1	z2
x2	z1

$\xrightarrow{\text{Remove duplicates}}$

$\Pi_{A,C}(R)$

A	C
x1	z1
x1	z2
x2	z1

PROJECT(R, X) $\Pi_X(R)$

SQL

```
select distinct last_name, first_name
from employees;
```

```
select last_name, first_name
from employees
group by last_name, first_name;
```

Usually needs a temporary table. Optimizations use indexes [1].

Relational algebra properties $\Pi_X\ (R)$

Rule 1: Project composition:

$$\Pi_{\{A_1,\dots,A_n\}}\left(\Pi_{\{B_1,\dots,B_m\}}(R)\right) = \Pi_{\{A_1,\dots,A_n\}}(R), \qquad \{A_1,\ \dots,\ A_n\} \subseteq \{B_1,\ \dots,\ B_m\}$$

```
select last_name, first_name, salary
from
  (select last_name, first_name, salary, job_id
   from employees);
```

```
select last_name, first_name, salary
from employees;
```

Select

- Unary operator
- Notations: SELECT(R, C) or $\sigma_C(R)$
- R is a relation; C is a logical formula with attributes of R, constants and operators: AND, OR, NOT, <, =, >, <=, >=, !=

- The result is a relation with all the attributes of R but with only those lines satisfying C.

SELECT(R, C) $\sigma_C(R)$

R

A	B	C
x1	y1	z1
x1	y1	z2
x1	y2	z2
x2	y2	z1

$$\sigma_{B='y2' \, or \, C='z2'}(R) \longrightarrow$$

$\sigma_C(R)$

A	B	C
x1	y1	z2
x1	y2	z2
x2	y2	z1

SELECT(R, C) $\sigma_C(R)$

SQL

```
select * from employees
where last_name = 'King' or first_name = 'Steven';
```

Optimizations use indexes.

Relational algebra properties $\sigma_C(R)$

Rule 2: Selection composition:

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

```
select job_id, job_title, min_salary, max_salary
from
    (select job_id, job_title, min_salary, max_salary
     from jobs
     where min_salary > 8000)
where min_salary < 10000;
```

```
select job_id, job_title, min_salary, max_salary
from jobs
where min_salary > 8000 and min_salary < 10000;
```

Relational algebra properties $\Pi_X(R)$ and $\sigma_C(R)$

Rule 3a: Selection and projection commute:

$$\Pi_{\{A_1,\ldots,A_n\}}(\sigma_C(R)) = \sigma_C\left(\Pi_{\{A_1,\ldots,A_n\}}(R)\right), \qquad C \text{ operands are in } \{A_1,\ldots,A_n\}$$

```
select job_title, min_salary
from
   (select job_id, job_title, min_salary, max_salary
    from jobs
    where min_salary > 8000);
```

```
select job_title, min_salary
 from
    (select job_title, min_salary
     from jobs)
where min_salary > 8000;
```

Relational algebra properties $\Pi_X(R)$ and $\sigma_C(R)$

Rule 3b: Selection and projection commute:

$$\Pi_{\{A_1,\ldots,A_n\}}(\sigma_C(R)) = \Pi_{\{A_1,\ldots,A_n,B_1,\ldots,B_m\}}(\sigma_C(\Pi_{\{A_1,\ldots,A_n,B_1,\ldots,B_m\}}(R))),$$

$$C\ operands\ are\ in\ \{A_1,\ldots,A_n,B_1,\ldots,B_m\}$$

```
select job_title, min_salary
from
   (select job_id, job_title, min_salary, max_salary
    from jobs
    where min_salary > 8000
          and max_salary <10000);
```

```
select job_title, min_salary
 from
     (select job_title, min_salary, max_salary
     from jobs
     where min_salary > 8000
           and max_salary <10000);
```

UNION

- Binary operator
- Notations: UNION(R, S) or $R \cup S$
- R and S relations; the result of union is the set of all tuples in R or S. --- - union on set of tuples.

- R and S must have compatibles relational schemas, i.e., the same number of attributes and the same attributes types.
- $R \cup S = S \cup R$

UNION(R, S) $R \cup S$

R

A	B	C
x1	y1	3
x1	y2	10
x2	y2	7

S

E	F	G
x1	z1	1
x1	z2	4
x1	y2	10
x2	z1	7

union,
remove
duplicates

$R \cup S$

A	B	C
x1	y1	3
x1	y2	10
x2	y2	7
x1	z1	1
x1	z2	4
x2	z1	7

UNION(R, S) $R \cup S$

SQL

```
select employee_id, start_date from job_history
union /*sorts the result*/
select employee_id, hire_date from employees;
```

```
select employee_id, start_date from job_history
union all  /* keeps duplicates */
select employee_id, hire_date from employees;
```

Optimizations: AVOID union, use temporary tables, see QueryOptimization.sql

UNION(R, S) $R \cup S$

SQL

```
select  id, uuid, random_number
from no_index
union
select  id, uuid, random_number
from no_index;

/* ~210sec
```

```
select  id, uuid, random_number
from no_index
union all
select  id, uuid, random_number
from no_index;

/* ~15sec
```

Optimizations: AVOID union, use temporary tables.

Relational algebra properties UNION and $\sigma_C(R)$

Rule 4: Selection and union commute:

$$\sigma_C(R_1 \cup R_2) = \sigma_C(R_1) \cup \sigma_C(R_1)$$

```
select employee_id, start_date from job_history
where employee_id > 110
union all
select employee_id, hire_date from employees
where employee_id > 110;
```

```
select employee_id, start_date from (
select employee_id, start_date from job_history
union all
select employee_id, hire_date from employees
)
where employee_id > 110; /*see exec. plans*/
```

Relational algebra properties UNION and $\Pi_X\ (R)$

Rule 5: Projection and union commute:

$$\Pi_{\{A_1,\dots,A_n\}}(R_1 \ \cup \ R_2) = \Pi_{\{A_1,\dots,A_n\}}(R_1) \cup \Pi_{\{A_1,\dots,A_n\}}(R_1)$$

```
select start_date from (
select employee_id, start_date from job_history
union all
select employee_id, hire_date from employees
);
```

```
select start_date from job_history
union all
select hire_date  from employees
```

DIFFERENCE

- Binary operator
- Notations: DIFFERENCE(R, S), MINUS(R,S) or $R - S$
- R and S relations; the result of difference is the set of all tuples in R that are not found in S.

  -- difference on set of tuples.
- R and S must have compatibles relational schemas, i.e., the same number of attributes and the same attributes types.

DIFFERENCE(R, S) $R - S$

R

A	B	C
x1	y1	3
x1	y2	10
x2	y2	7

S

E	F	G
x1	z1	1
x1	z2	4
x1	y2	10
x2	z1	7

R minus S →

$R - S$

A	B	C
x1	y1	3
x2	y2	7

DIFFERENCE(R, S) $R - S$

SQL

```
select department_id
from departments
minus
select department_id
from employees;
```

```
select department_id
from departments d
where d.department_id not in
   (select department_id
    from employees
    where department_id is not null);
```

Optimizations: *not in* or *not exists*.

Relational algebra properties UNION and $R - S$

Rule 6: Selection and difference commute:

$$\sigma_C(R_1 - R_2) = \sigma_C(R_1) - \sigma_C(R_1)$$

```
select department_id
from departments
where department_id > 120
minus
select department_id
from employees
where department_id > 120
```

```
select * from (
  select department_id
  from departments
    minus
 select department_id
 from employees
)
where department_id > 120; /*see exec. plans*/
```

INTERSECT

- Binary operator
- Notations: INTERSECT(R, S) or $R \cap S$
- R and S relations; the result of intersection is the set of all tuples that are both in R and in S. --- intersection on set of tuples.
- R and S must have compatibles relational schemas, i.e., the same number of attributes and the same attributes types.
- $R \cap S = S \cap R$
- $R \cap S = R - (R - S) = S - (S - R)$

INTERSECT(R, S) $R \cap S$

R

A	B	C
x1	y1	3
x1	y2	10
x2	y1	7

S

E	F	G
x1	z1	1
x1	z2	4
x1	y2	10
x2	y1	7

intersect →

$R \cup S$

A	B	C
x1	y2	10
x2	y1	7

INTERSECT(R, S) $R \cap S$

SQL

```
select department_id
from employees
intersect
select department_id
from job_history;
```

Optimizations: *in* or *exists*.

DIVISION

- Binary operator
- Notations: DIVISION(R, S) or $R \div S$
- R and S relations; the result of difference is the set of all tuples to which any of the tuples in S can be added to obtain a tuple in R

- $R \div S = \{t^{n-m} | \forall s \in S,\ (t, s) \in R\}$

- If R has n attributes and S has m < n attribute the result of DIVISION has n – m attributes.

DIVISION(R, S) $R \div S$

R

X	Y	Q	T
x1	y1	q1	t1
x1	y1	q2	t2
x2	y2	q1	t1
x3	y3	q2	t2
x4	y4	q1	t1
x4	y4	q2	t2

S

Q	T
q1	t1
q2	t2

R ÷ S →

$R \div S$

Q	T
x1	y1
x4	y4

DIVISION(R, S) $R \div S$

x1, y1 are "associated" with all tuples in S (R is the "associative" table):

R

X	Y	Q	T
x1	y1	q1	t1
x1	y1	q2	t2
x2	y2	q1	t1
x3	y3	q2	t2
x4	y4	q1	t1
x4	y4	q2	t2

S

Q	T
q1	t1
q2	t2

R ÷ S →

$R \div S$

Q	T		
x1	y1	q1	t1
x4	y4		

$$R \div S = \{t^{n-m} | \forall s \in S,\ (t, s) \in R\}$$

$$t = (x1, y1); s = (q1, t1)\ (x1, y1, q1, t1) \in R$$

DIVISION(R, S) $R \div S$

x1, y1 are associated with all tuples in S (R is the "associative" table):

R

X	Y	Q	T
x1	y1	q1	t1
x1	y1	q2	t2
x2	y2	q1	t1
x3	y3	q2	t2
x4	y4	q1	t1
x4	y4	q2	t2

S

Q	T
q1	t1
q2	t2

R ÷ S

$R \div S$

Q	T		
x1	y1	q2	t2
x4	y4		

$$R \div S = \{t^{n-m} | \forall s \in S,\ (t,s) \in R\}$$

$$t = (x1, y1); s = (q2, t2)\ \ (x1, y1, q2, t2) \in R$$

DIVISION(R, S) $R \div S$

x1, y1 are "associated" with all tuples in S (R is the "associative" table):

R

X	Y	Q	T
x1	y1	q1	t1
x1	y1	q2	t2
x2	y2	q1	t1
x3	y3	q2	t2
x4	y4	q1	t1
x4	y4	q2	t2

S

Q	T
q1	t1
q2	t2

R ÷ S

$R \div S$

Q	T		
x1	y1	q2	t2
x4	y4		

$$R \div S = \{t^{n-m} | \forall s \in S,\ (t,s) \in R\}$$

$$\boldsymbol{\nexists s \text{ such as } (t,s) \notin R}$$

DIVISION(R, S) $R \div S$

(x2, y2) is not "associated" with all tuples in S :

R

X	Y	Q	T
x1	y1	q1	t1
x1	y1	q2	t2
x2	y2	q1	t1
x3	y3	q2	t2
x4	y4	q1	t1
x4	y4	q2	t2

S

Q	T
q1	t1
q2	t2

R ÷ S →

$R \div S$

Q	T		
x1	y1		
x4	y4		
~~x2~~	~~y2~~	q2	t2

DIVISION(R, S) $SQL\ NOT\ EXISTS/COUNT$

Code for employees working on all projects with a budget < 10000

R

EMPLOYEE_ID	PROJECT_ID
125	1
136	2
125	2
200	2
148	1
148	2
200	3
148	3

S

PROJECT_ID
2
3

$R \xrightarrow{R \div S}$

$R \div S$

EMPLOYEE_ID
200
148

S = PROJECT(SELECT(PROJECTS, budget < 10000) , project_id)
R = PROJECT(WORKS_ON, project_id, employee_id)

DIVISION(R, S) $SQL\ NOT\ EXISTS/COUNT$

Code for customers buying tickets to all comedies < 10000

R

CUSTOMER_ID	MOVIE_ID
100	10
101	7
101	10
100	5
102	1
101	2
102	5
100	7
102	10

S

MOVIE_ID
5
7
10

R ÷ S →

$R \div S$

CUSTOMER_ID

S = PROJECT(SELECT(MOVIES, genre = comedies) , movie_id)
R = PROJECT(BUY, customer_id, movie_id)

PRODUCT

- Binary operator
- Notations: PRODUCT(R, S) or $R \times S$
- R and S relations; the result of difference is the set of all tuples with *m + n* attributes, first *m* attributes form a tuple of R, last *n* attributes form a tuple of S

- $R \times S = \{(t, s) | t \in R, s \in S\}$

PRODUCT(R, S) $R \times S$

R

X	Y	Q
x1	y1	q1
x2	y2	q2
x3	y3	q3
x4	y4	q4

S

U	V
u1	v1
u2	v2

$\xrightarrow{R \times S}$

$R \times S$

X	Y	Q	U	V
x1	y1	q1	u1	v1
x2	y2	q2	u1	v1
x3	y3	q3	u1	v1
x4	y4	q4	u1	v1
x1	y1	q1	u2	v2
x2	y2	q2	u2	v2
x3	y3	q3	u2	v2
x4	y4	q4	u2	v2

PRODUCT(R, S) $R \times S$

SQL

```
select e.*, d.*
from employees e, departments d ;

select e.*, d.*
from employees e cross join departments d ;
```

JOIN

- Binary operator
- Notations: JOIN(R, S)

Natural Join:

$$\text{JOIN}\ (R, S) = \Pi_{\{i_1,\ldots,\, i_m\}}\left(\sigma_{R.A1=S.A1\ \wedge\ \ldots R.An=S.An}(R\ \times S)\right)$$

A1, … An are the intersection of attributes of R and S, i1, … im is the union of attributes of R and S minus A1, … An.

JOIN

- Binary operator
- Notations: JOIN(R, S)

Semi Join:

$$\text{SEMIJOIN }(R, S) = \Pi_{\{r_1, \dots, r_m\}}(\text{JOIN }(R, S))$$

r1, … rm is the union of attributes of R.

JOIN

- Binary operator
- Notations: JOIN(R, S)

Θ join:

$$\Theta\text{ - JOIN }(R, S, \sigma_{cond}) = \sigma_{cond}(R \times S)$$

Relational algebra properties $\Pi_X\ (R)$ and $\sigma_C(R)$

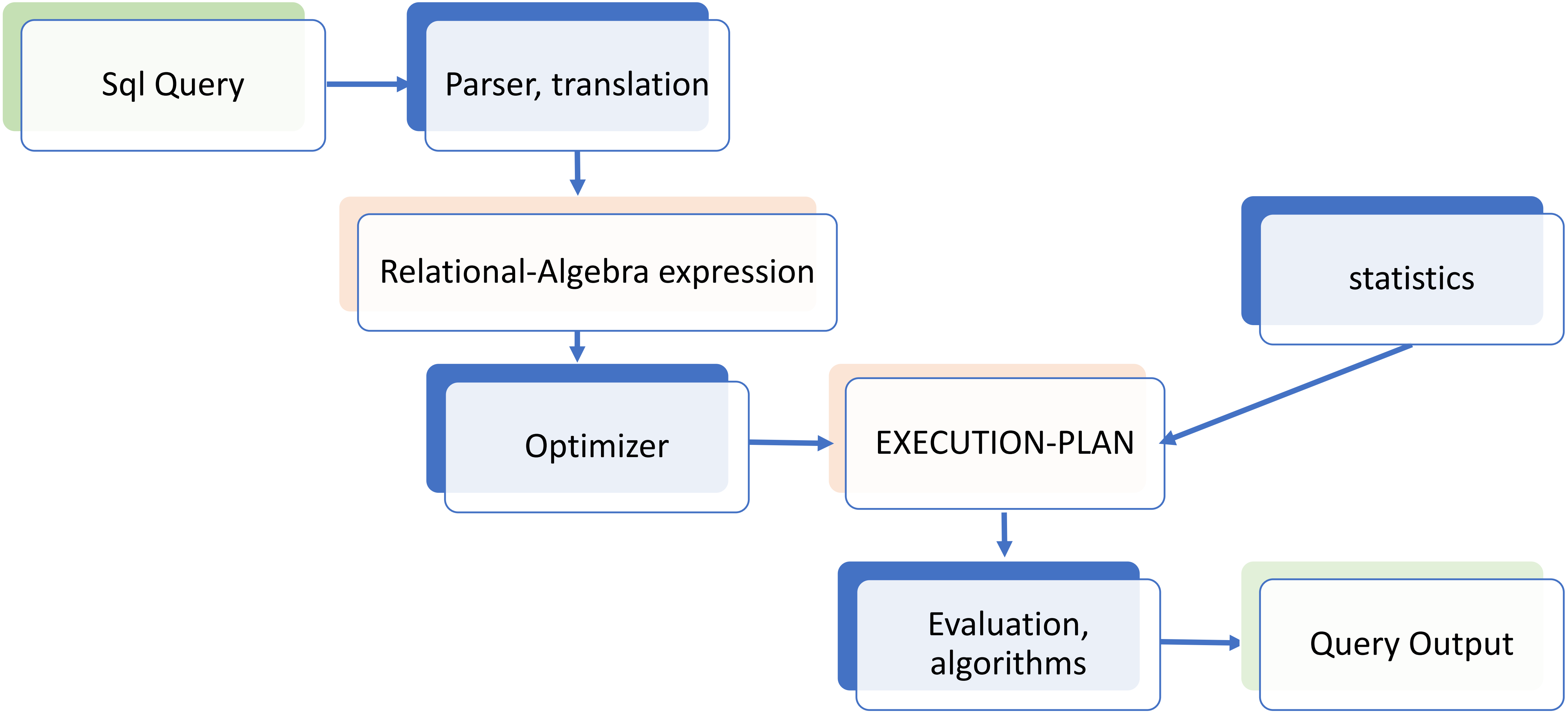
Rule 6: JOIN/PRODUCT commute:

$$JOIN(R,S) = JOIN(S,R),$$
$$R \times S \qquad = S \times R$$

```
select last_name, department_name
from employees e join departments d
    on (e.department_id  = d.department_id);
```

```
select last_name, department_name
from departments d join employees e
    on (e.department_id  = d.department_id)
```

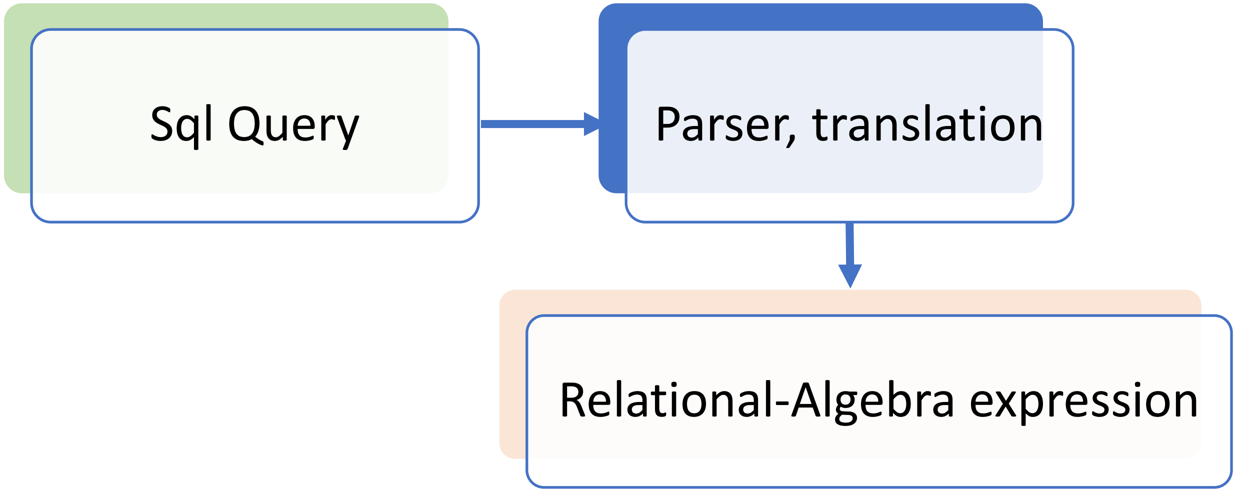
Query execution


Sql Query
Parser, translation
Relational-Algebra expression
statistics
Optimizer
EXECUTION-PLAN
Evaluation, algorithms
Query Output

Sql Query → Parser, translation

```
select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
        on p1.prod_min_price = p2.prod_min_price
```

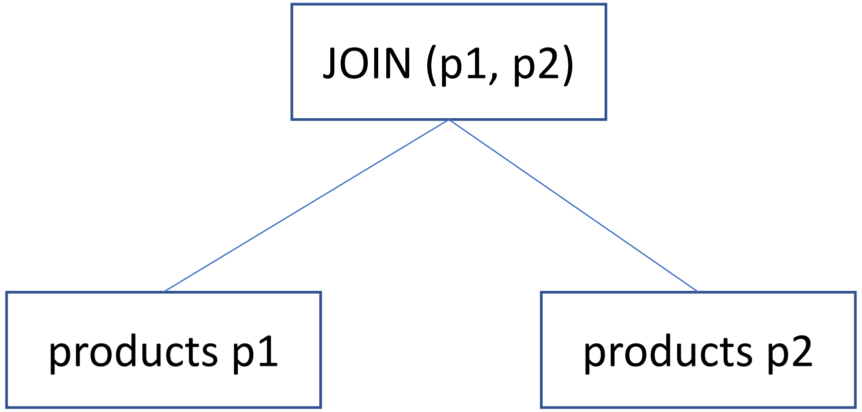
check syntax, table names, view names, column names

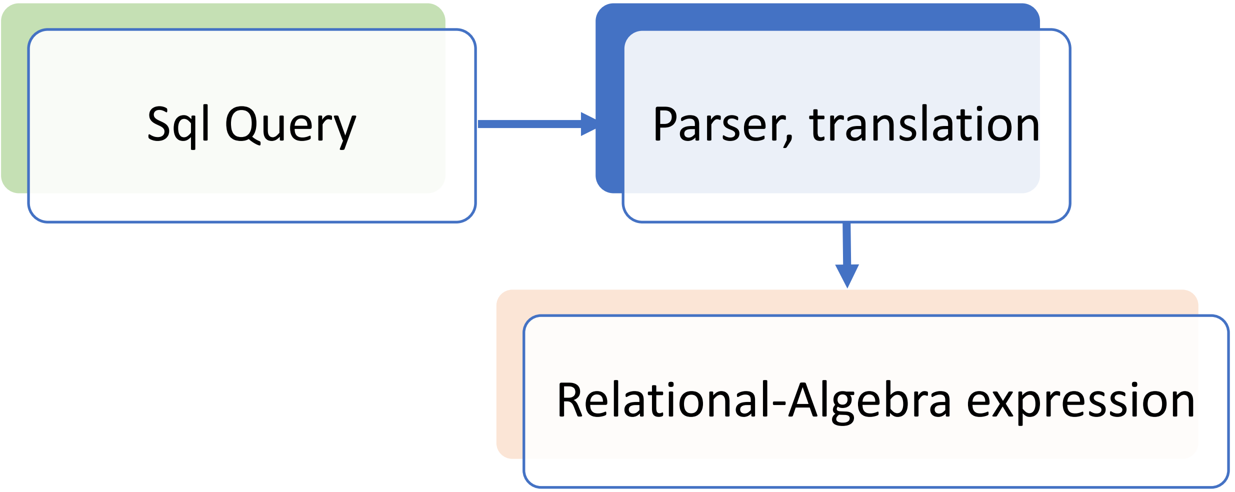



```
select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
        on p1.prod_min_price = p2.prod_min_price
```

relations + operators

$$JOIN(\,p1, p2)$$

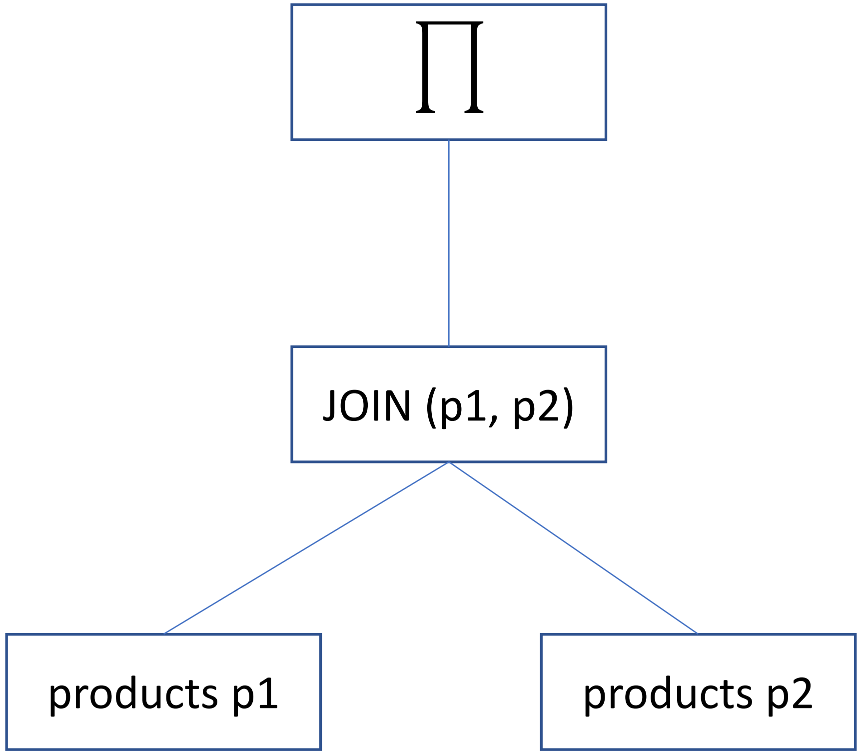


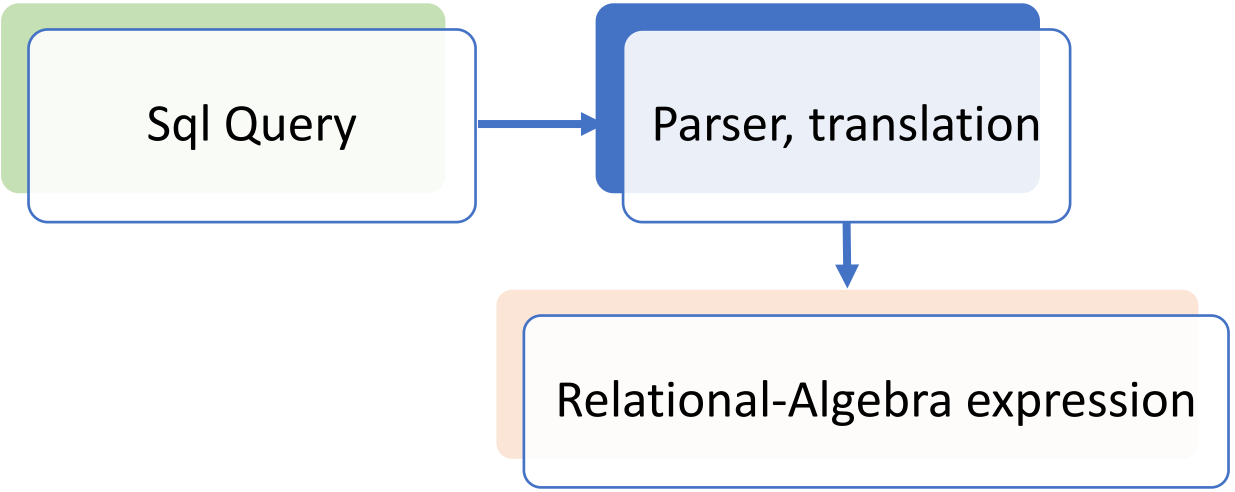


select p1.prod_name, p2.prod_name, p1.min_price
from products p1 join products p2
on p1.prod_min_price = p2.prod_min_price

relations + operators

$$\prod_{p1.prodname, p2.prodname,\, p1.minprice} JOIN(\,p1, p2)$$

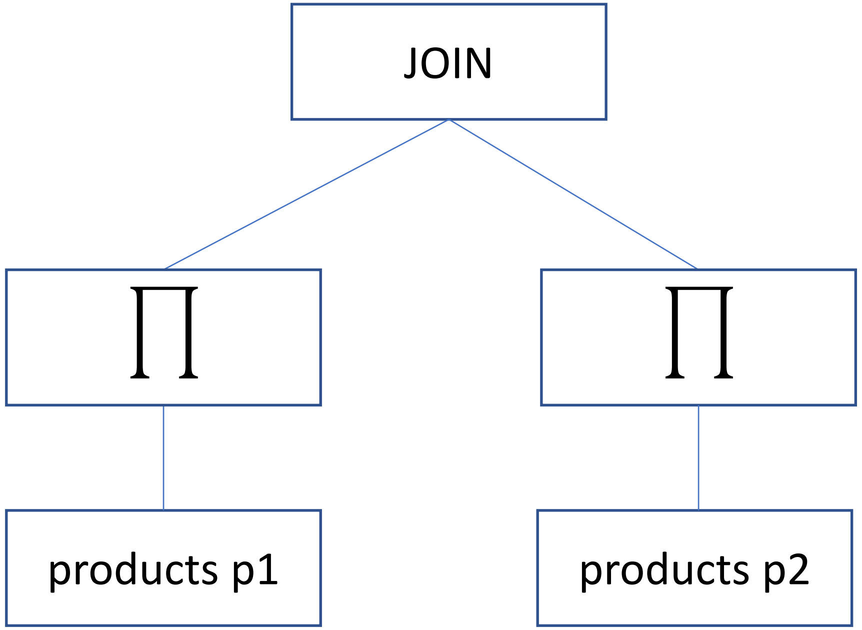


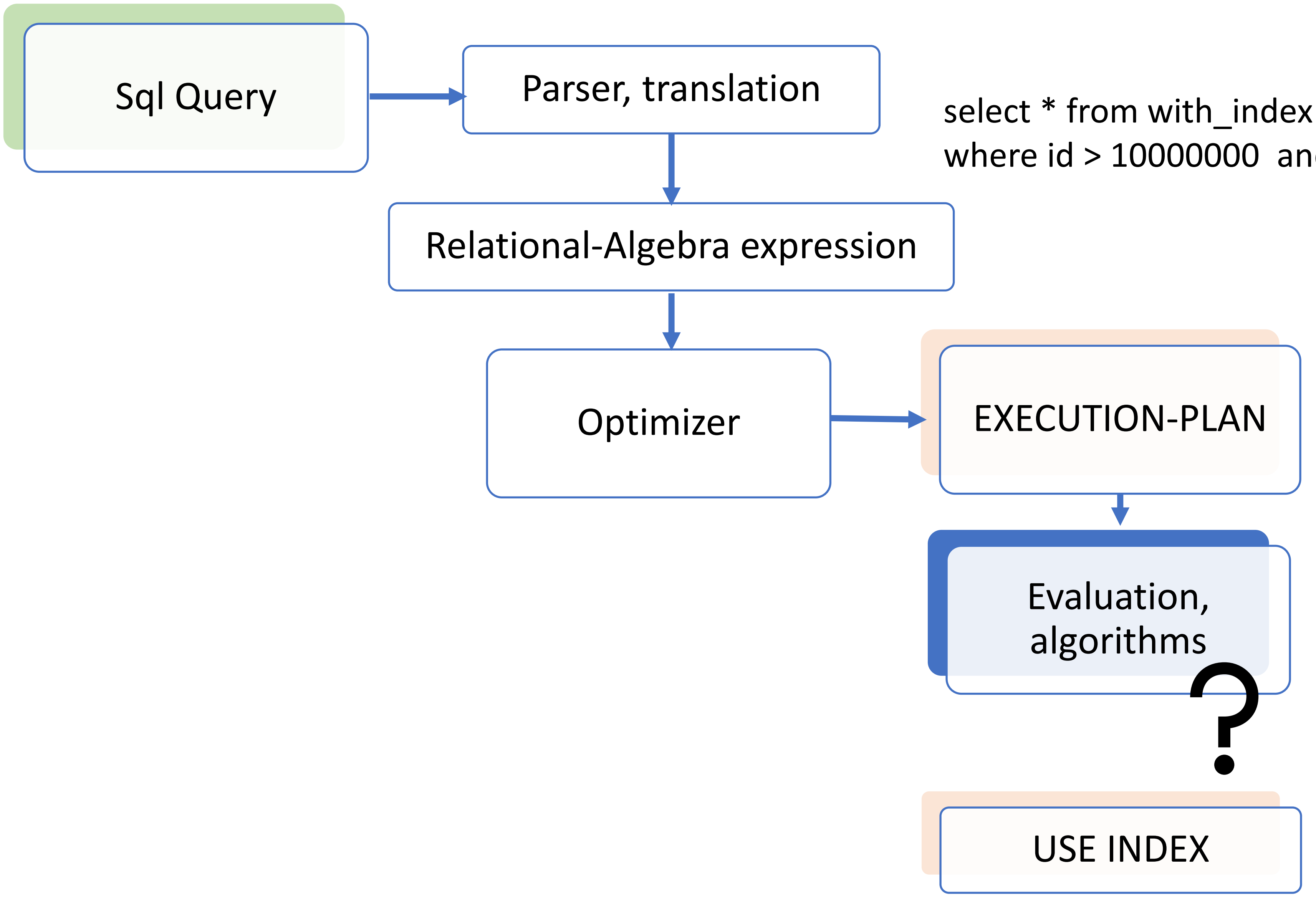


select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
on p1.prod_min_price = p2.prod_min_price

relations + operators

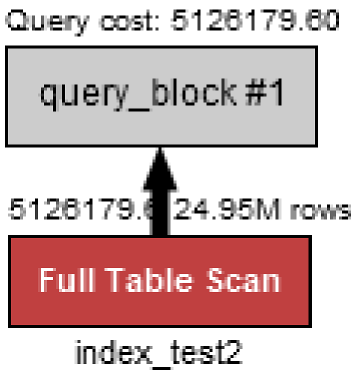
$$JOIN(\prod_{name,minprice} p1, \prod_{name,minprice} p2)$$



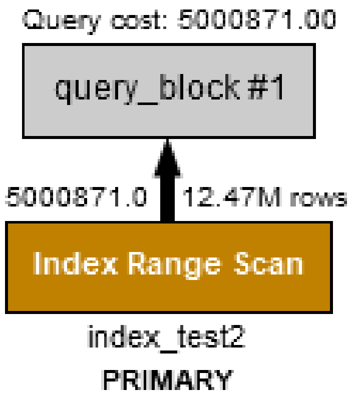


select * from with_index
where id > 10000000 and id < 20000000;

select * from with_index
where random_number > '1111111111111' ;



select * from with_index
where id > 10000000 and id < 20000000;

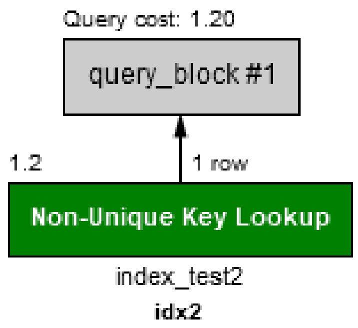


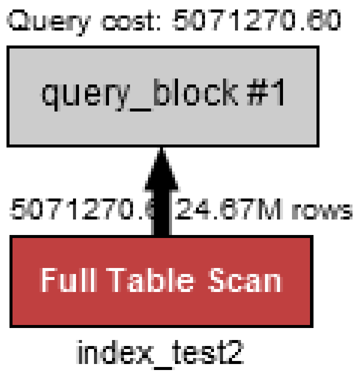
?

USE INDEX

```
select * from with_index
where random_number = '1111111111111';
```

```
select * from with_index
where random_number > '1111111111111’
and random_number < '2222222222222';
```





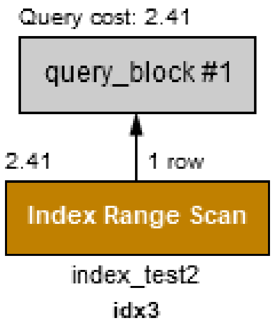
?

USE INDEX

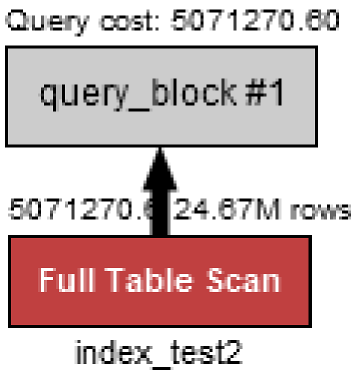
alter table with_index
add random_number2 varchar(10);

update with_index
set random_numer2 = random_number
where id % 1250000 = 0;

select * from with_index
where random_number > '1111111111111’
and random_number < '222222222222';



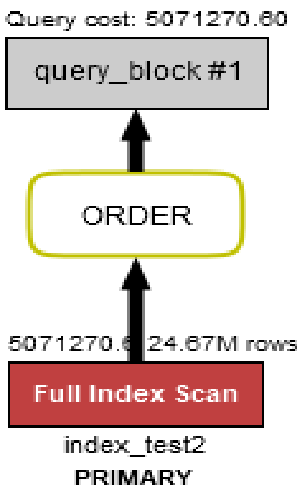
select * from with_index
where random_number > '1111111111111’
and random_number < '222222222222';



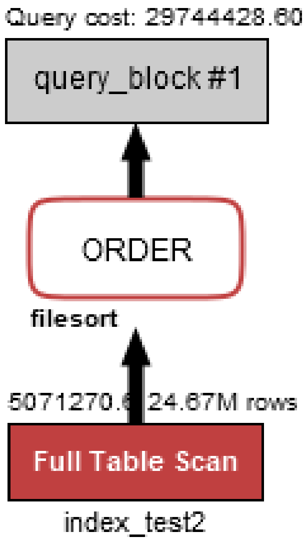
?

USE INDEX

select * from with_index order by id;

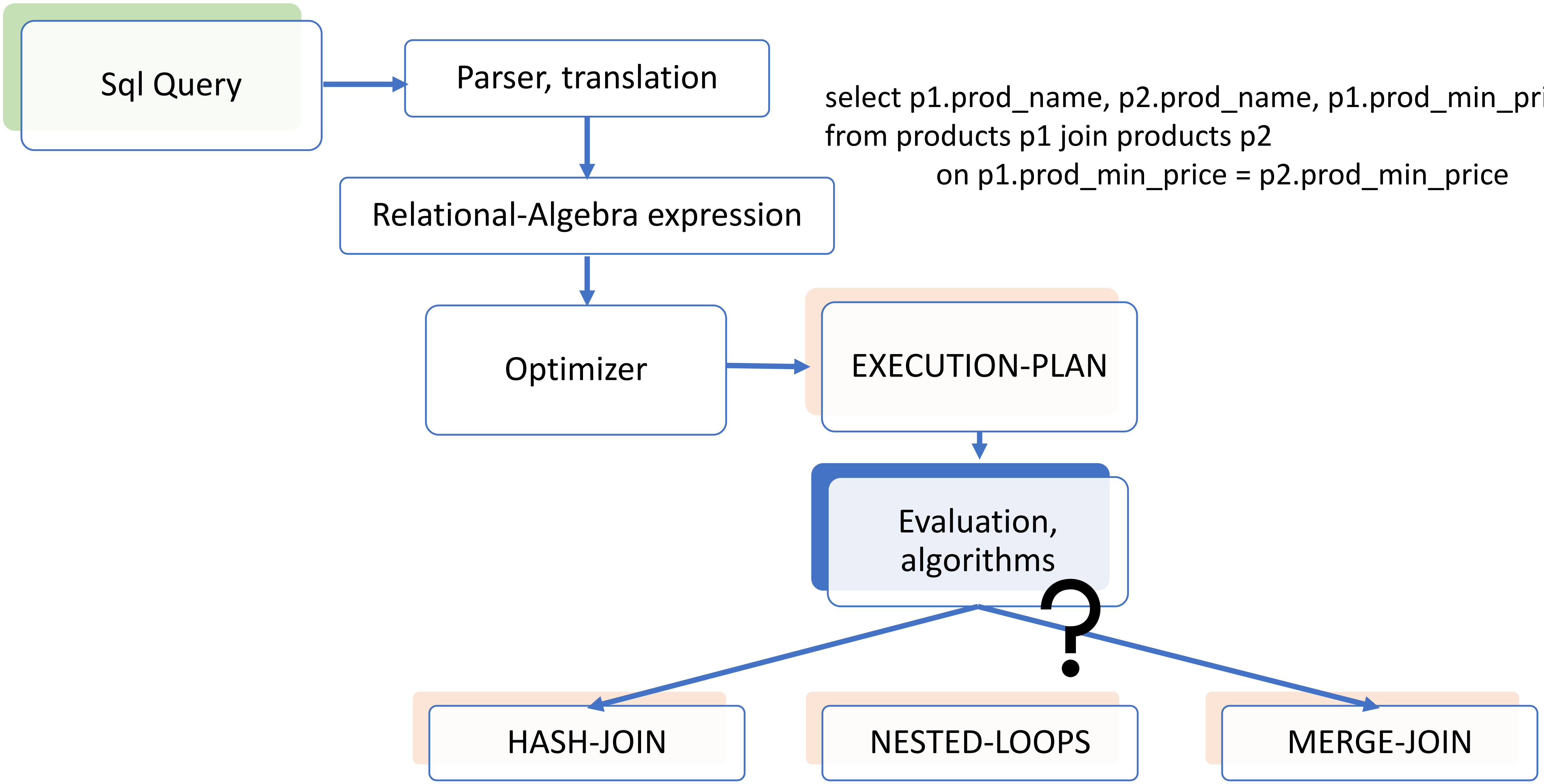


select * from with_index order by random_number2;

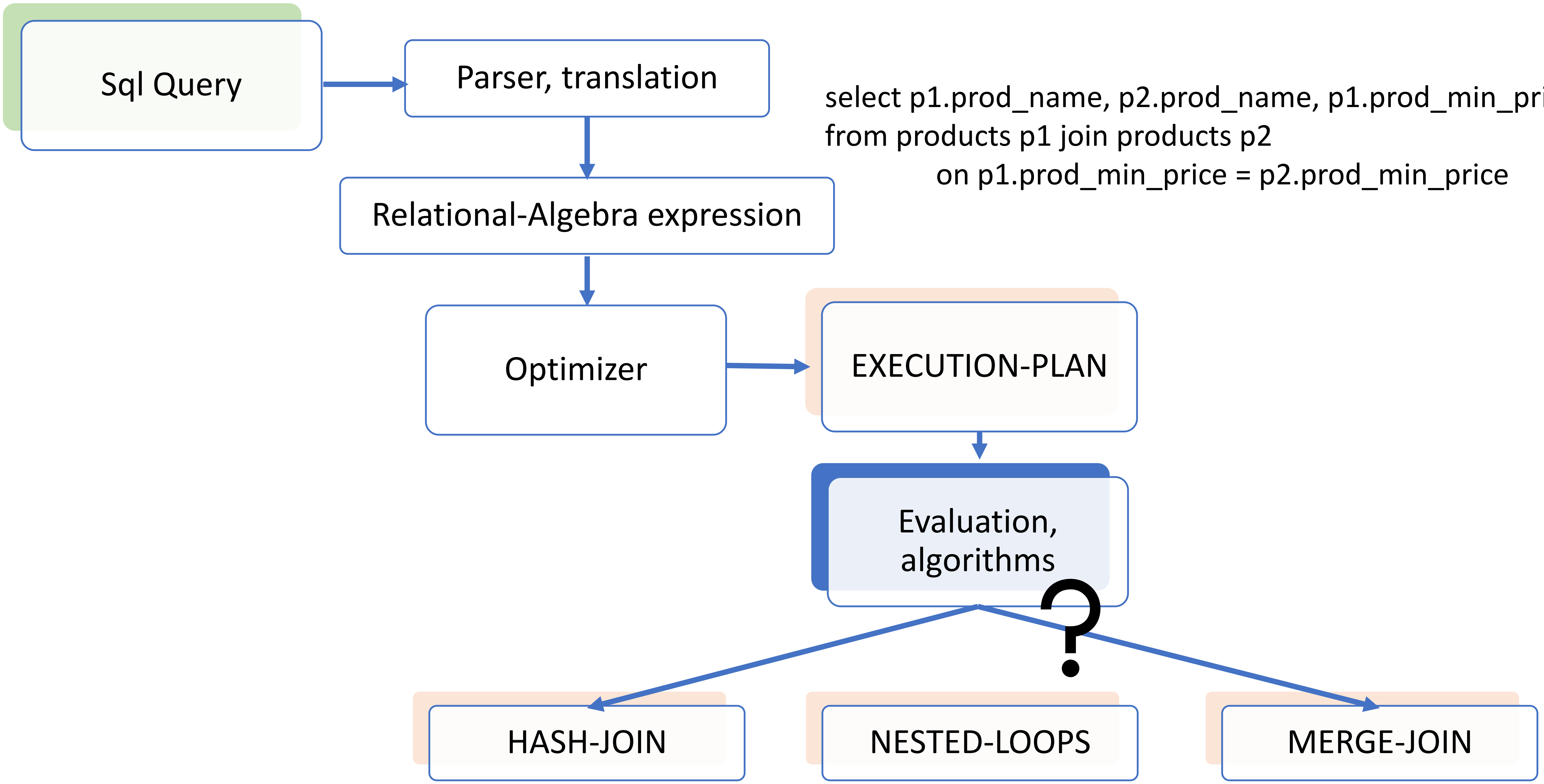


?

USE INDEX




```
select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
        on p1.prod_min_price = p2.prod_min_price
```




```
select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
        on p1.prod_min_price = p2.prod_min_price
```

NESTED-LOOPS

```
for each tuple t_r in r
        for each tuple t_s in s
                if join condition θ for pair (t_r , t_s) = true
                        result = result ∪ (t_r , t_s);
        end
end
```

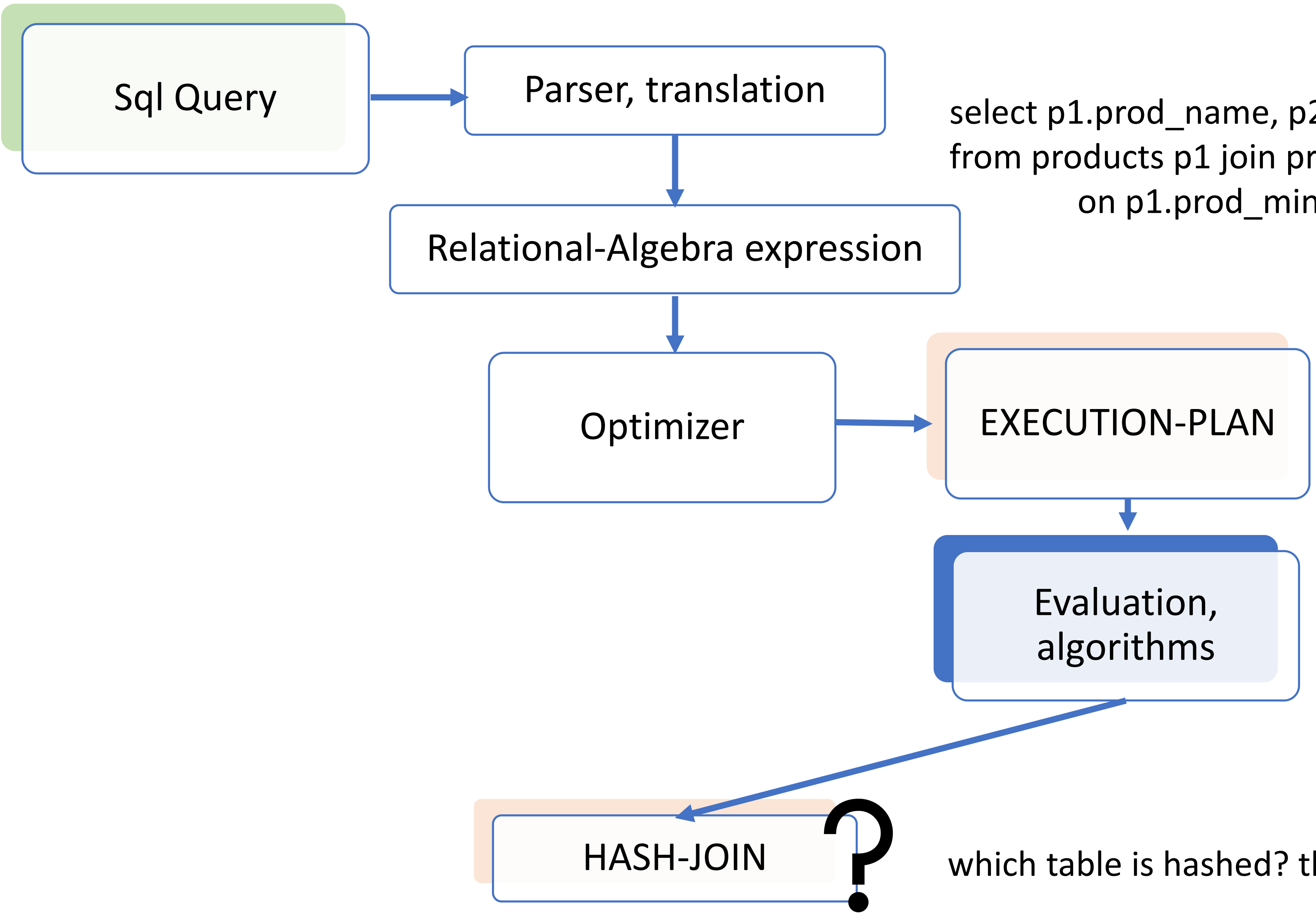
- Optimizations:
  - Block nested loops: each block in s is loaded in memory once for each block on r.
  - Index nested loops: if an index is present on s, it can be used to search for tuples satisfying join condition.

MERGE-JOIN

```
sort R
sort S
r = R.first
s = S.first
while r <> nill and s <> nill
    if r.id > s.id
        s = next tuple in S
    else if r.id < s.id
        r = next tuple in R
    else
        result = result ∪ (r,s)
    r’ = next tuple in R
    while r’<>nill and r’.id = s.id
        result = result ∪ (r’,s)
    s’ = next tuple in S
    while s’<>nill and r.id = s’.id
        result = result ∪ (r,s’)
    r = next tuple in R
    s = next tuple in S
```

HASH-JOIN

partition R based on hash(key) -> $Rn_1,\ldots,Rn_p$
partition S based on hash(key) -> $Sn_1,\ldots,Sn_p$
for each n_{key}
    build in-memory hash index
        for Rn_{key} -partition
    for each tuple t_s in Sn_{key} -partition
        probe t_s , add matching tuples




```
select p1.prod_name, p2.prod_name, p1.prod_min_price
from products p1 join products p2
        on p1.prod_min_price = p2.prod_min_price
```

which table is hashed? the “smaller”

Bloom filters

Bloom filters

- Probabilistic data structure, check membership for a value in a set.

- How it works: S, set of n values → *const * n* bits
  calculate hash(v) ∈ $[1, const * n]$
  set bit hash(v) to 1

  Test w ∈ S → hash(w) = 1 ?

- Small probability of **false positive**.
  w1 ∈ S, w2 ∉ S hash(w1) = hash(w2)

Bloom filters

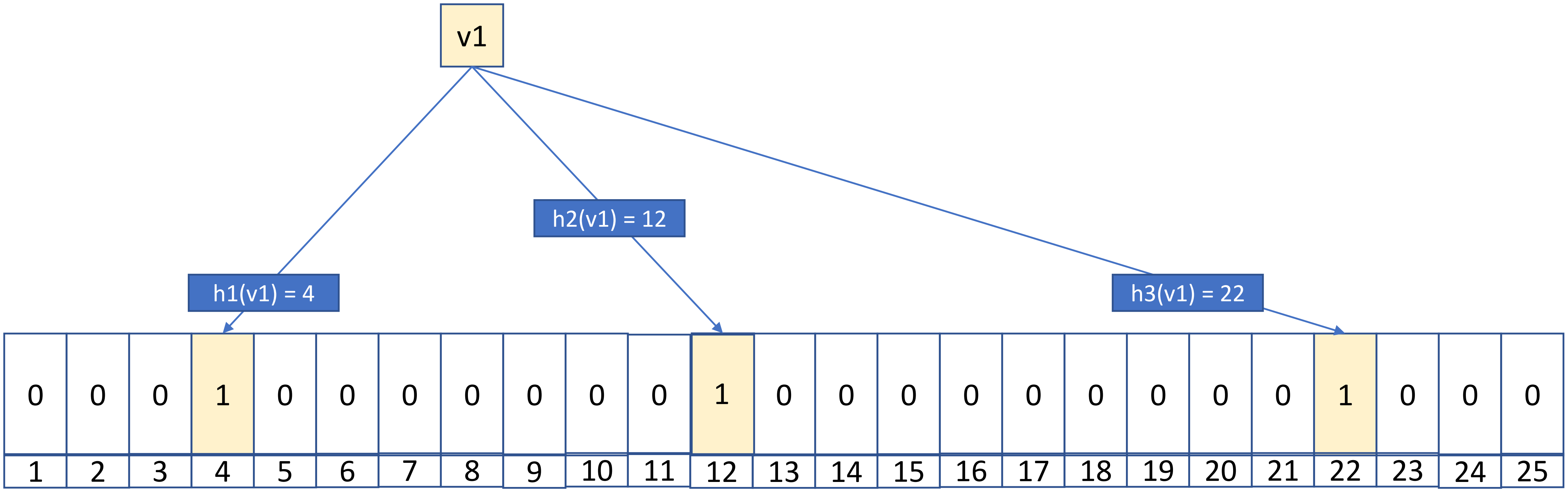
- To reduce the probability of false positives, use k > 1 independent hash functions.

- How it works: S, set of n values → *const * n* bits

  calculate $h_1(v)$, $h_2(v)$ … $h_k(v) \in [1, const * n]$

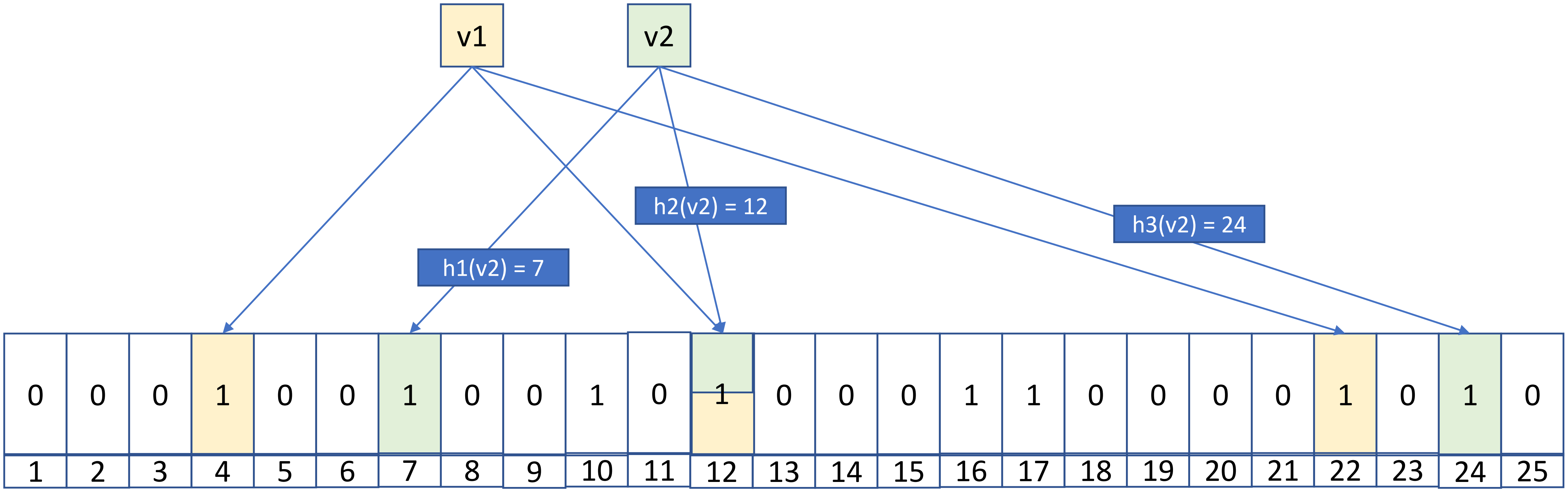
  set bits $h_1(v)$, $h_2(v)$ … $h_k(v)$ to 1

  Test $w \in S$ → $h_1(v) = 1$ and $h_2(v) = 1$ … and $h_k(v) = 1$?



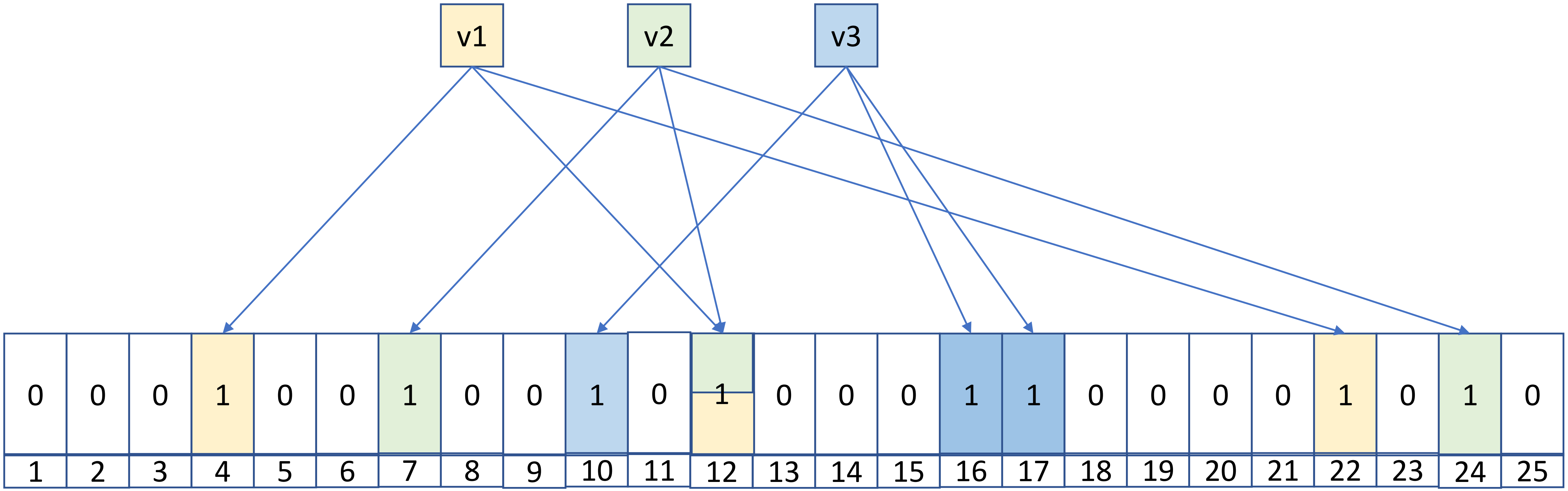
Small probability of **false positive**.

Probability of **false negative** = 0.



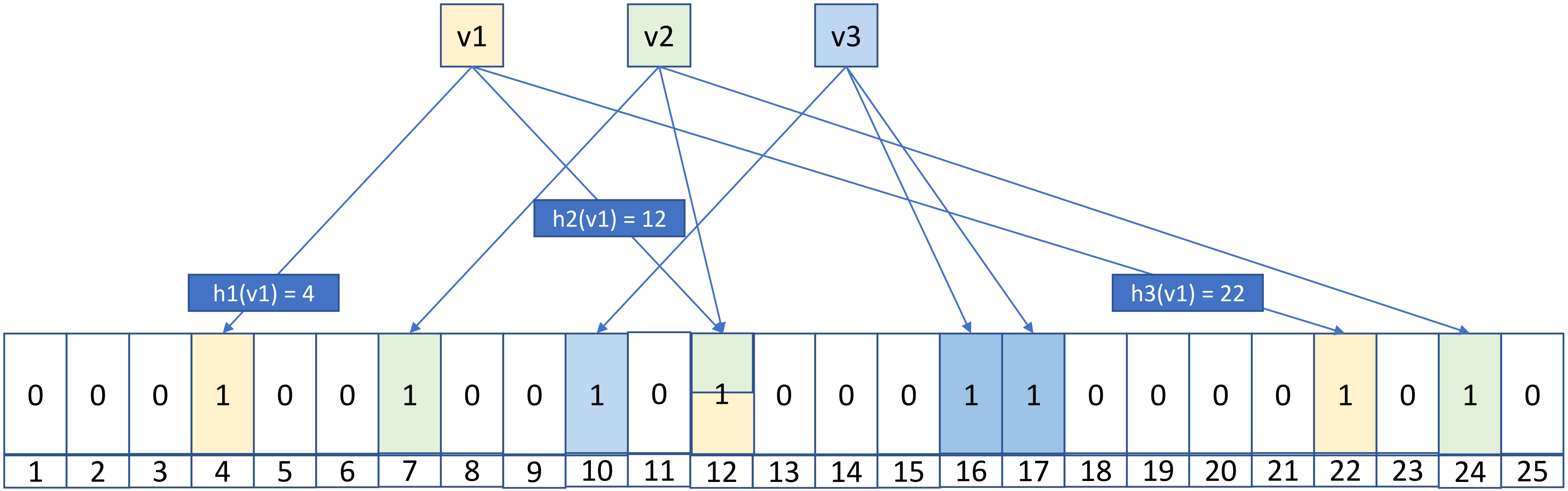
Small probability of **false positive**.

Probability of **false negative** = 0.



Small probability of **false positive**.

Probability of **false negative** = 0.

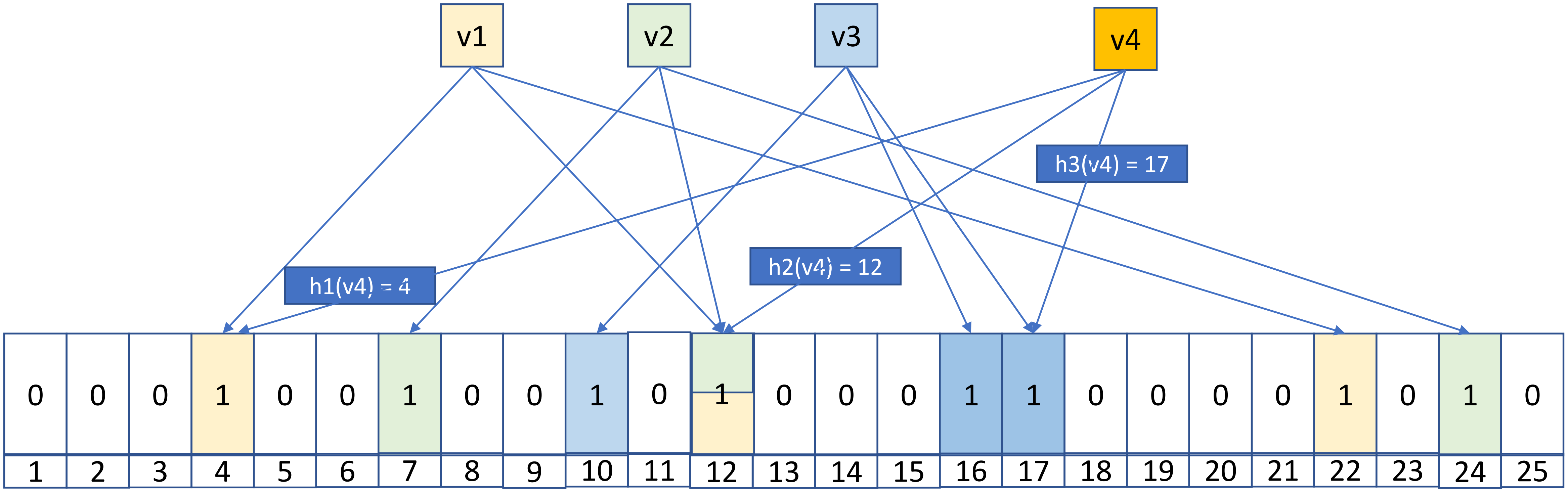


Small probability of **false positive**.

Probability of **false negative** = 0.

false positive. Value *w: B[h1(w)] = 1 B[h2(w)] =1 … B[hk[w]] = 1*

Each hash of w equals a hash of an element in the set



Small probability of **false positive**.

Probability of **false negative** = 0.

false positive. Value *w: B[h1(w)] = 1 B[h2(w)] =1 … B[hk[w]] = 1*

Each hash of w equals a hash of an element in the set

Bloom filters

- Used only to add elements or the test membership.

- Once an element is added to the filter it cannot be removed. Why?

- If all bits are set to 1, the probability of false positives increases.

  More space → more accuracy.
- More hash functions

  Latency → more accuracy.

Bloom filters – independent hashing

- A family of hash functions $H = \{h: U \to [1..m]\}$ is k-independent if $\forall\ (x_1, x_2 \ldots x_k) \in U^k$ and $\forall\ (y_1, y_2 \ldots y_k) \in [1..m]^k$:
  - $Pr_{h \in H}\,[h(x_1) = y_1 \wedge h(x_2) = y_2 \ldots \wedge h(x_k) = y_k] = \frac{1}{m^k}$
- $h(x_1)$ uniformly distributed.
- $h(x_1), h(x_2)$, … $h(x_k)$ independent random variables.

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \quad \text{or}$$

$$P = \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

- m = 10 * n and k = 7 $\simeq$ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

h(w) != h1(v1)

$$P = \left(1 - \boxed{\left(1 - \frac{1}{m}\right)}^{kn}\right)^{k} \quad \text{or}$$

- m = 10 * n and k = 7 ≃ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(1 - \boxed{\left(1 - \frac{1}{m}\right)^{kn}}\right)^{k} \quad \text{or}$$

$h_1(w)$!= $h_1(v1)$
$h_1(w)$!= $h_2(v1)$
.....
$h_1(w)$!= $h_k(v1)$
$h_1(w)$!= $h_1(v2)$
$h_1(w)$!= $h_2(v2)$
...
$h_1(w)$!= $h_k(v2)$
...
$h_1(w)$!= $h_k(vn)$

- m = 10 * n and k = 7 ≃ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(\boxed{1 - \left(1 - \frac{1}{m}\right)^{kn}}\right)^{k} \quad \text{or}$$

$h_1(w) = h_1(v1)$
or
$h_1(w) = h_2(v1)$
or
.....
$h_1(w) = h_k(v1)$
or
$h_1(w) = h_1(v2)$
...
or
$h_1(w) = h_k(vn)$

- m = 10 * n and k = 7 $\simeq$ 0,01

General optimization rules

General optimization rules

- Execute selections first
  - Reduce relation size (number of rows)

- Avoid cross-joins, use joins

- First join to be executed is the one obtaining the smaller relation

- Execute projections first

Estimating Query Cost

rule-based execution plans/optimization

obsolite

cost-based execution plans

IO-cost

CPU-cost

IO cost

CPU time

number of blocks transferred from storage - b

number of random I/O accesses - s

cost for processing a tuple

cost for processing an index entry

$b * t_T + s * t_S$

cost for processing comparison operators

cost for processing a function .....

Estimating cost

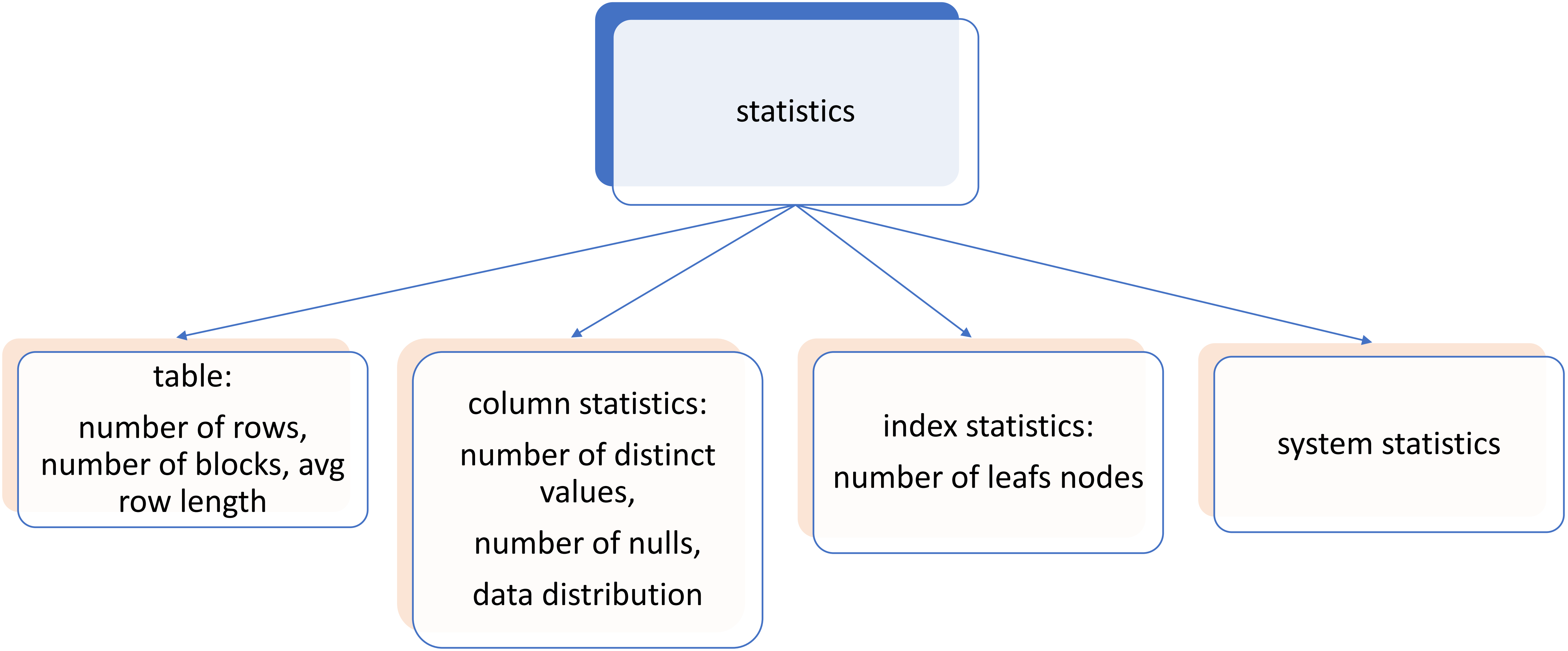
- Linear search

	COST	OBSERVATIONS
Linear search	b * t_T + t_S	Search for initial block Transfer r blocks
Linear search B-tree, Equality on key	(h_i + 1) * (t_T + t_S)	h_i height of the index, each operation requires a seek and a block transfer.
Linear search clustering B-tree, Equality on non-key	h_i * (t_T + t_S)+ b * tT + tS	b blocks storing the specified search key are stored sequentially
Linear search secondary B-tree, Equality on non-key	(h_i + n) * (t_T + t_S)	n number of records fetched; each record may be on a different block.

Estimating cost - example

- Linear search

	COST	OBSERVATIONS
Linear search	b * t_T + t_S	Search for initial block Transfer r blocks
Linear search B-tree, Equality on key	(h_i + 1) * (t_T + t_S)	h_i height of the index, each operation requires a seek and a block transfer.
Linear search clustering B-tree, Equality on non-key	h_i * (t_T + t_S)+ b * tT + tS	b blocks storing the specified search key are stored sequentially
Linear search secondary B-tree, Equality on non-key	(h_i + n) * (t_T + t_S)	n number of records fetched; each record may be on a different block.


statistics
table:
number of rows, number of blocks, avg row length
column statistics:
number of distinct values,
number of nulls,
data distribution
index statistics:
number of leafs nodes
system statistics

- [1] https://dev.mysql.com/doc/refman/8.0/en/group-by-optimization.html
- [2] https://computing.derby.ac.uk/c/codds-twelve-rules/
- [3] https://antognini.ch/papers/BloomFilters20080620.pdf