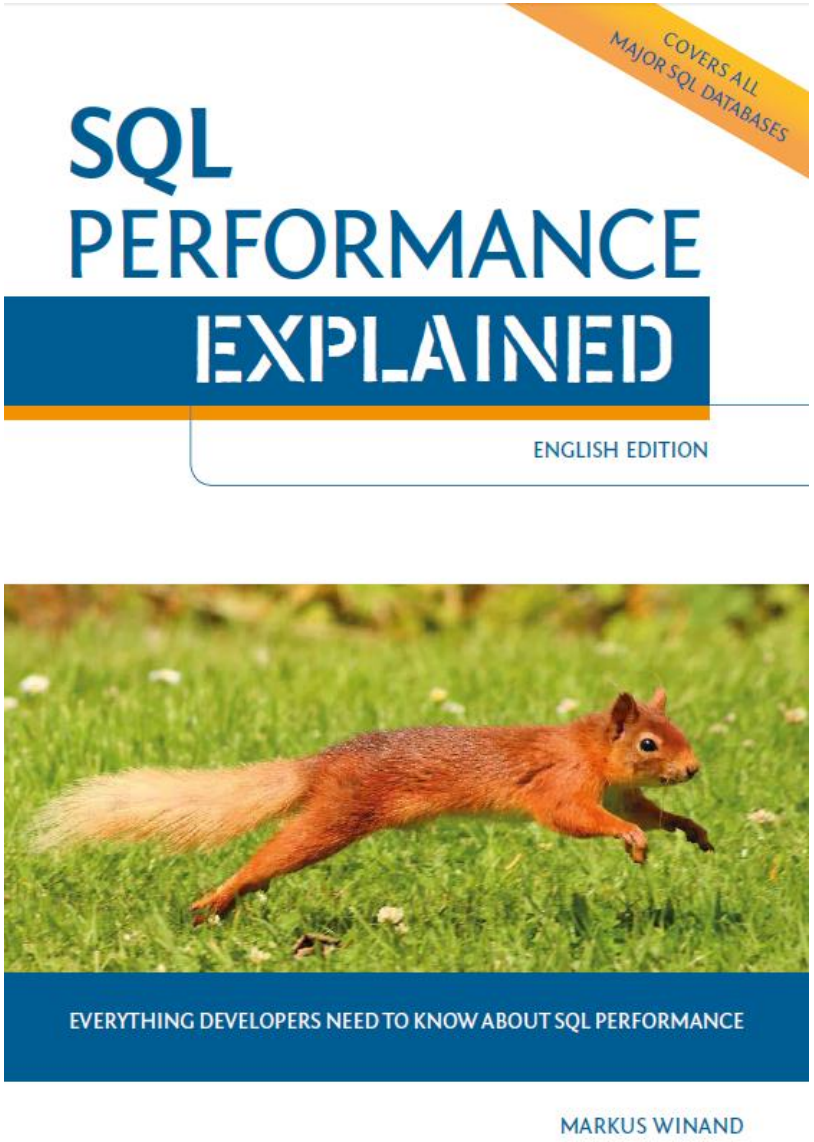
Practica SGBD

http://use-the-index-luke.com/



Sintactic & Semantic

- Putem considera o interogare SQL ca fiind o propozitie din engleza ce ne indica ce trebuie facut fara a ne spune cum este facut:

```
SELECT date_of_birth
    FROM employees
    WHERE last_name = 'WINAND'
```

La baza unei aplicatii ce nu merge stau doua greseli umane*

- Autorului unei interogari SQL nu ii pasa (de obicei) ce se intampla “in spate”.
- Autorul interogarii nu se considera vinovat daca timpul de raspuns al SGBD-ului este mare (evident, cel care l-a inventat nu prea a stiut ce face).
- Solutia ? Simplu: nu mai folosim Oracle, trecem pe MySQL, PostgreSQL sau SQL Server.

*Una dintre ele este de a da vina pe calculator.

De fapt…

- Singurul lucru pe care dezvoltatorii trebuie sa il invete este cum sa indexeze corect.
- Cea mai importanta informatie este felul in care aplicatia va utiliza datele.
- Traseul datelor nu este cunoscut nici de client, nici de administratorul bazei de date si nici de consultantii externi; singurul care stie acest lucru este dezvoltatorul aplicatiei !

...cuprins....

- Anatomia unui index
- Clauza WHERE
- Performanta si Scalabilitate
- JOIN
- Clustering
- Sortare & grupare
- Rezultate partiale
- INSERT, UPDATE, DELETE

Anatomia unui index

- *"An index makes the query fast"* - cat de rapid?

```
-------------------------------------------------------------------------
   | Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)|
   |   0 | SELECT STATEMENT  |               |    10 |   330 |     2   (0)|
   |*  1 |  INDEX RANGE SCAN| NUME_PRENUME  |    10 |   330 |     2   (0)|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("FIRST_NAME"='WINAND')
```

Anatomia unui index

- *“An index makes the query fast”*

```
-----------------------------------------------------------------------
   | Id  | Operation          | Name         | Rows  | Bytes | Cost (%CPU)|
   |   0| SELECT STATEMENT   |              |    10 |   330 |     4   (0)|
   | *  1|  TABLE ACCESS FULL| EMPLOYEES    |    10 |   330 |     4   (0)|
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("FIRST_NAME"='WINAND')
```

Anatomia unui index

- Un index este o structura* distincta intr-o baza de date ce poate fi construita utilizand comanda **`create index`**.
- Are nevoie de propriul spatiu pe HDD si pointeaza tot catre informatiile aflate in baza de date (la fel ca si cuprinsul unei carti, redundanta pana la un anumit nivel – sau chiar 100% redundant: SQL Server sau MySQL cu InnoDB folosesc Index-Organized Tables [IOT]).

* vom detalia pana la un anumit nivel (nu complet)

Anatomia unui index

- Cautarea dupa un index este asemanatoare cu cautarea intr-o carte de telefon.
- Indexul din BD trebuie sa fie mult mai optimizat din cauza dinamicitatii unei BD

  [insert / update / delete]
- Indexul trebuie mentinut fara a muta cantitate mare de informatie.

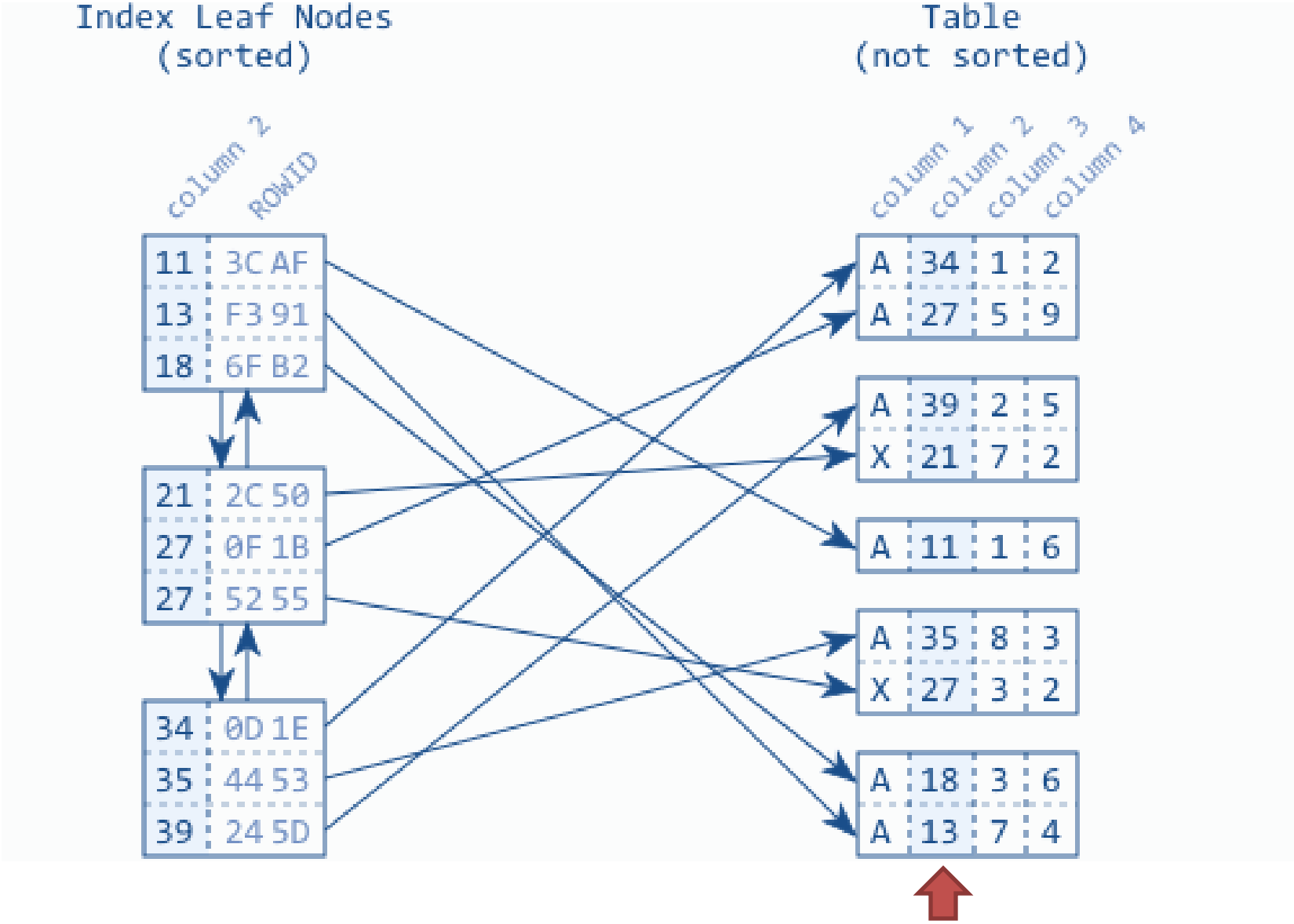
Anatomia unui index

- Cum functioneaza ?
  - pe baza unei liste dublu inlantuite
  - pe baza unui arbore de cautare
- Prin intermediul listei se pot insera cantitati mari de date fara a fi nevoie sa le deranjam pe cele existente.
- Arborele este utilizat pentru a cauta datele indexate (B-trees)

Anatomia unui index

- Bazele de date utilizeaza listele pentru a inlantui “frunzele” arborelui mentionat
- Fiecare bloc al unui index are o aceeasi dimensiune, contine cate un pointer catre urmatoarea frunza si unul catre o linie a unui tabel.

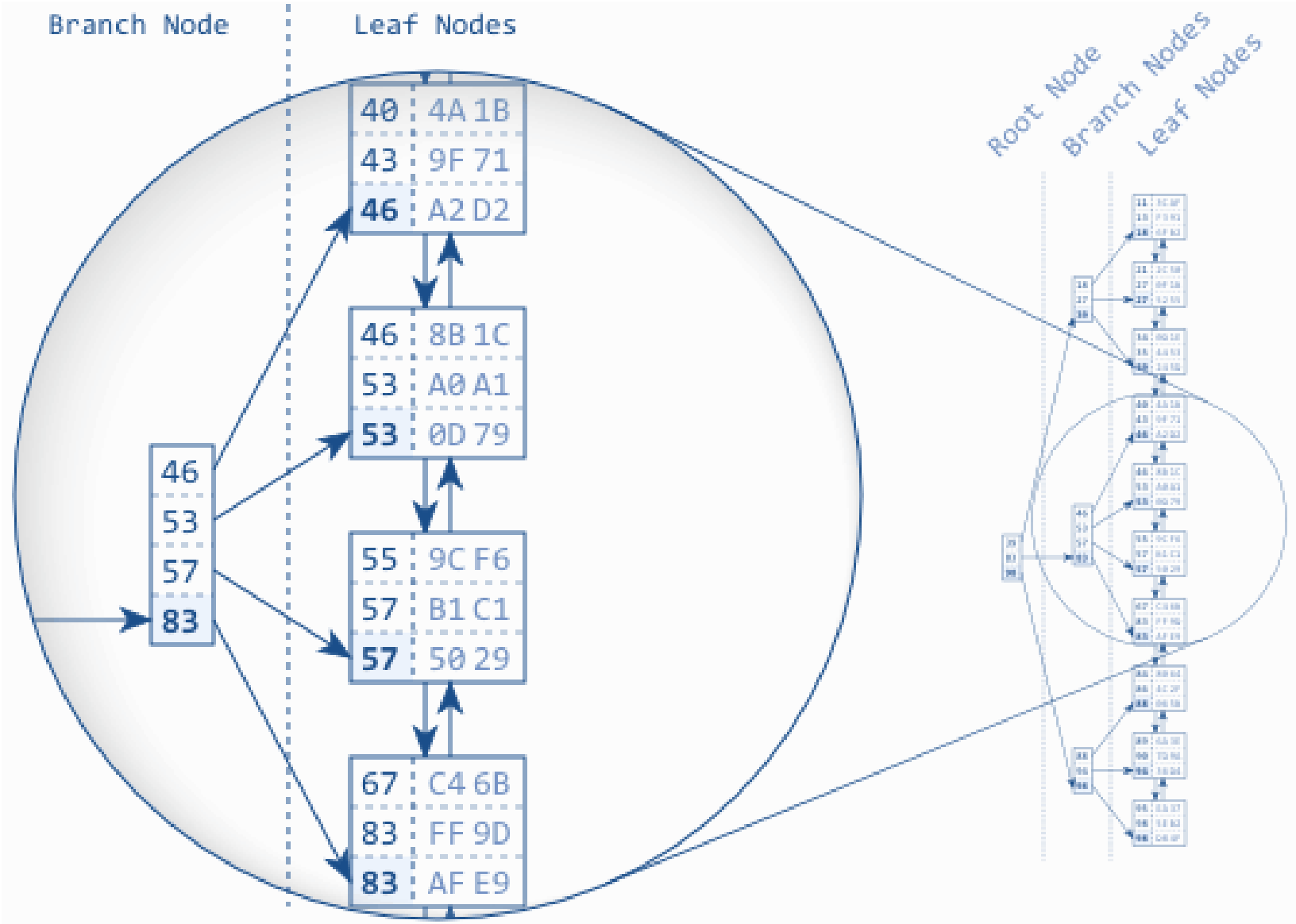
Anatomia unui index



Anatomia unui index

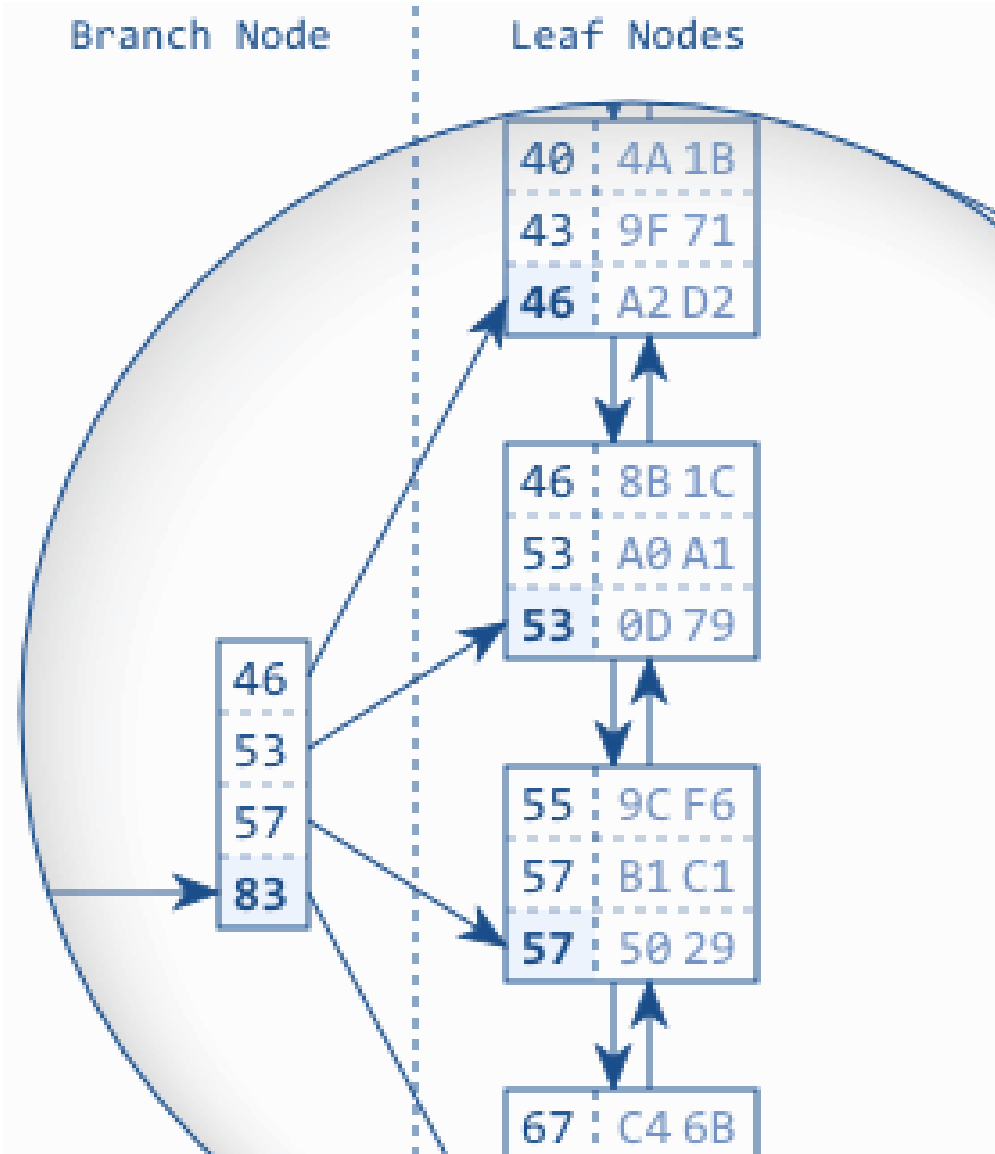
- “frunzele” nu sunt stocate pe disc in ordine sau avand o aceeasi distributie – pozitia pe disk nu corespunde cu ordinea logica a indecsilor (de exemplu, daca indexam mai multe numere intre 1 si 100 nu e neaparat ca 50 sa se afle exact la mijloc).
- SGBD-ul are nevoie de cea de-a doua structura pentru a cauta rapid intre indecsii amestecati.

Anatomia unui index

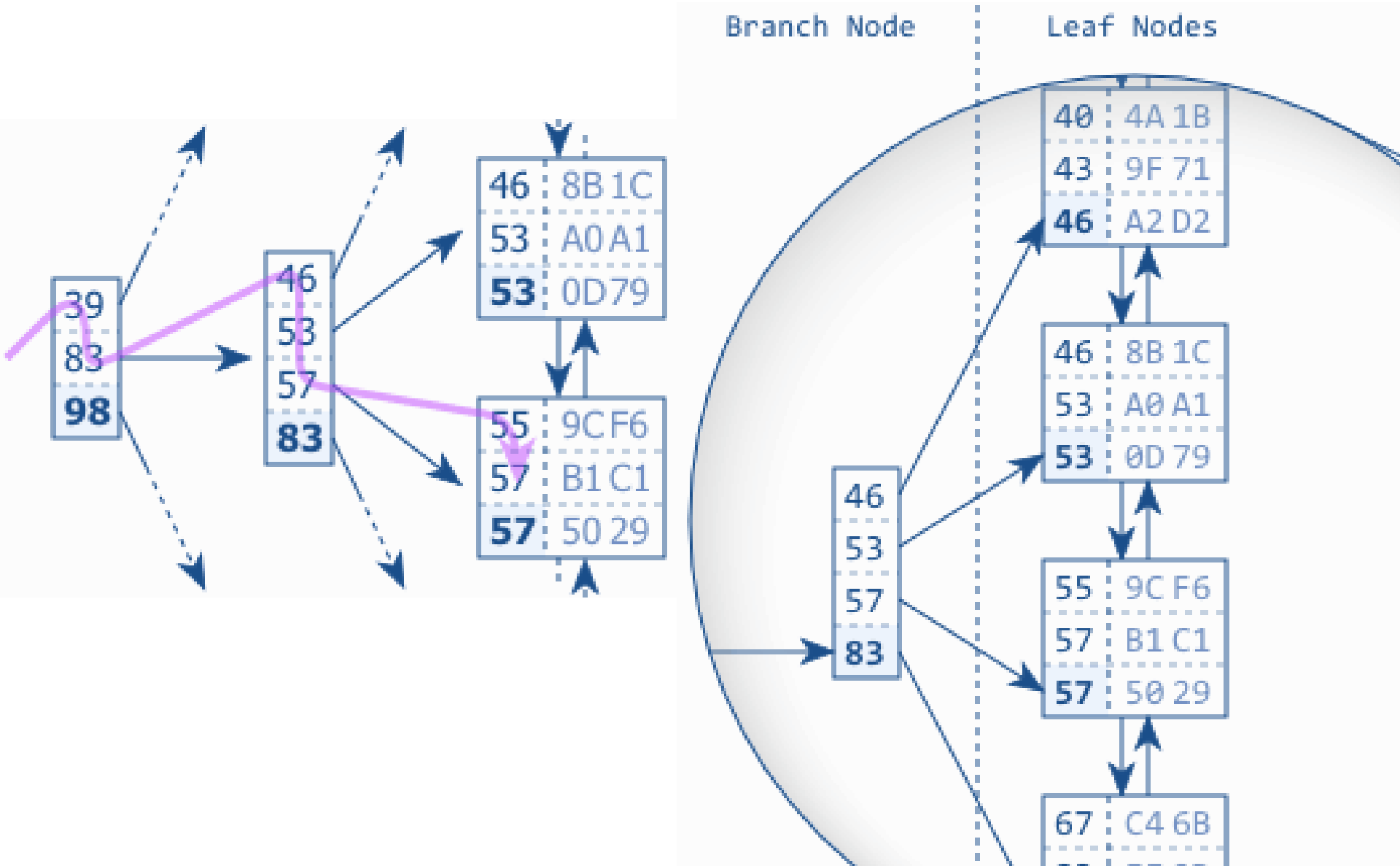


Anatomia unui index

Pointerul catre urmatorul nivel indica cea mai mare valoare a acestui urmator nivel.



Anatomia unui index



Anatomia unui index

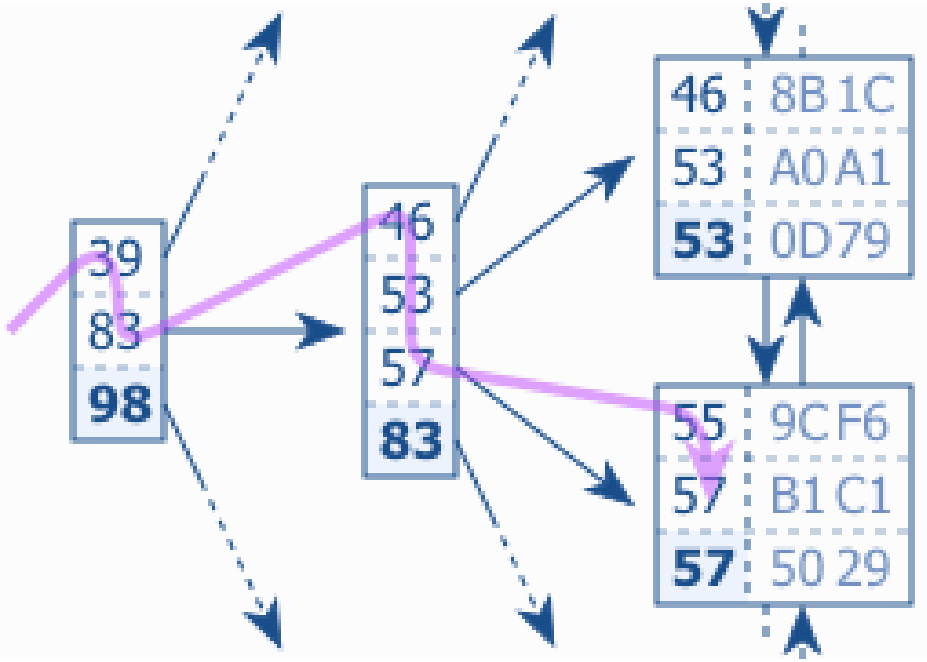
- Un *B-tree* este un arbore echilibrat, nu un arbore binar !
- Adancimea arborelui este identica spre oricare dintre frunze.
- Odata creat, baza de date mentine indecsii in mod automat, indiferent de operatia efectuata asupra bazei de date (insert/delete/update)
- *B-tree*-ul faciliteaza accesul la o frunza;
- Cat de repede ? [*first power of indexing*]

Anatomia unui index

- Desi gasirea informatiei se face in timp logaritmic, exista “mitul” ca un index poate degenera (si ca solutie este “reconstruirea indexului”). - fals deoarece arborele se autobalanseaza.

Anatomia unui index

- De ce ar functiona un index greu ?



- Atunci cand sunt mai multe randuri (57,57…)

Anatomia unui index

- De ce ar functiona un index greu ?
- Dupa gasirea indexului corespunzator, trebuie obtinut randul din tabela

- Cautarea unei inregistrari indexate se face in 3 pasi:
  - Traversarea arborelui [limita superioara: adancimea arborelui: oarecum rapid]
  - Cautarea frunzei in lista dublu inlantuita [incet]
  - Obtinerea informatiei din tabel [incet]

Anatomia unui index

- Este o conceptie gresita sa credem ca arborele s-a dezechilibrat si de asta cautarea este inceata. In fapt, traversarea arborelui pare sa fie cea mai rapida.
- Dezvoltatorul poate “intreba” baza de date despre felul in care ii este procesata interogarea.

Anatomia unui index

- In Oracle:

  INDEX UNIQUE SCAN

  INDEX RANGE SCAN

  TABLE ACCESS BY INDEX ROWID

- Cea mai costisitoare este INDEX RANGE SCAN.
- Daca sunt mai multe randuri, pentru fiecare dintre ele va face TABLE ACCESS – in cazul in care tabela este imprastiata in diverse zone ale HDD, si aceasta operatie devine greoaie.

Planul de executie

- Pentru a interoga felul in care Oracle proceseaza o interogare: **EXPLAIN PLAN FOR**

```
SQL> EXPLAIN PLAN FOR select * from emp;

Explained.
```

- Pentru a afisa rezultatul, se executa

```
select * from table(dbms_xplan.display);
```

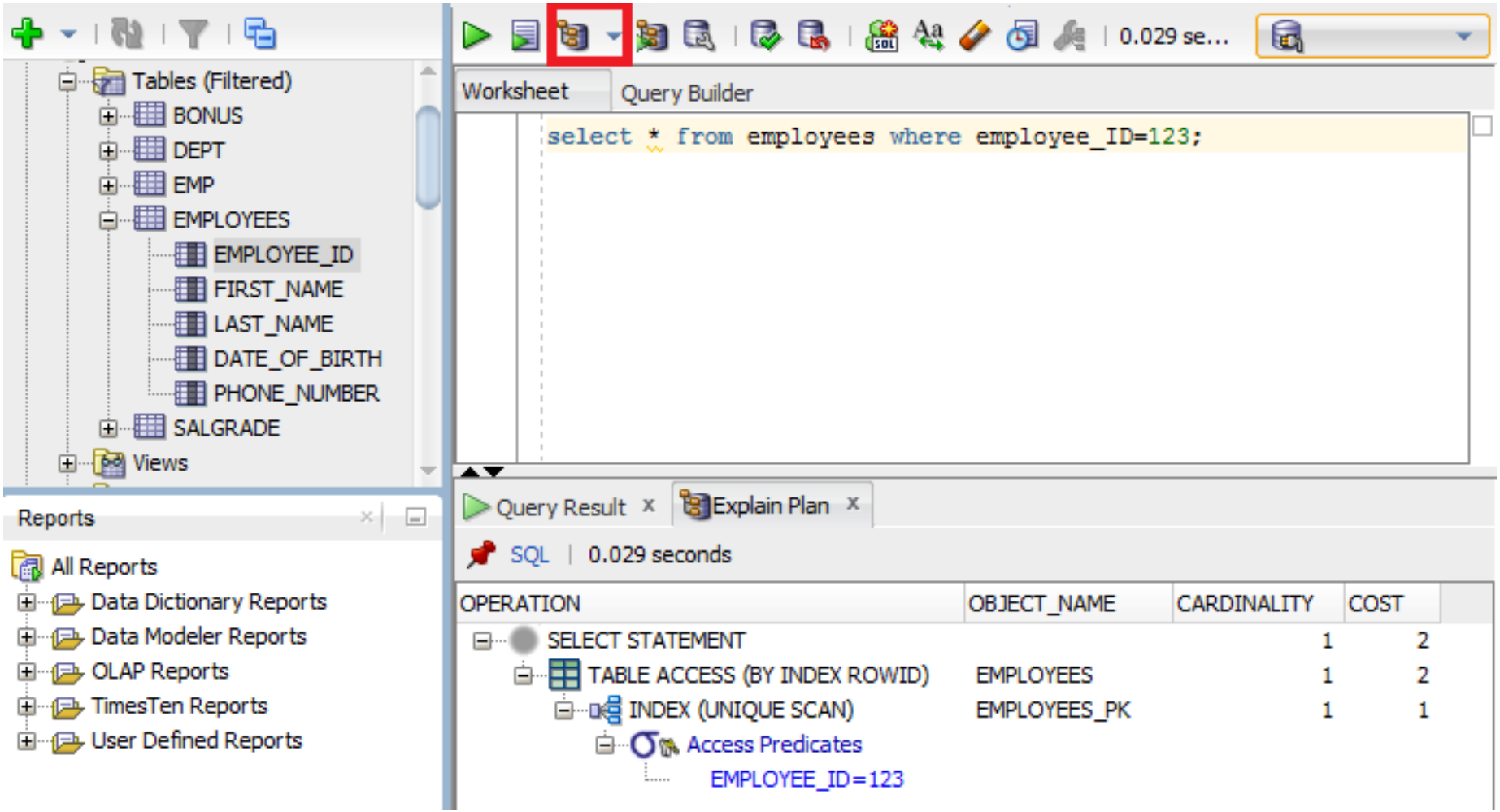
Planul de executie

```
SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3956160932

-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    16 |   704 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |    16 |   704 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------

8 rows selected.
```

Planul de executie

```
SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
Plan hash value: 2886526025

-------------------------------------------------------------------------------------
| Id  | Operation                    | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |     1 |    44 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP   |     1 |    44 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | ENAME |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ENAME"='JONNY')

14 rows selected.
```

Sau, versiunea SQL Developer…



Clauza WHERE

Clauza **WHERE**

- Clauza **WHERE** dintr-un select defineste conditiile de cautare dintr-o interogare SQL si din acest motiv este nucleul interogarii – din acest motiv influenteaza cel mai puternic rapiditatea cu care sunt obtinute datele
- Chiar daca **WHERE** este cel mai mare dusman al vitezei, de multe ori este “aruncat” doar “pentru ca putem”
- Un **WHERE** scris rau este principalul motiv al vitezei de raspuns a BD.

Clauza **WHERE**

```
CREATE TABLE employees (
   employee_id NUMBER NOT NULL,
   first_name VARCHAR2(1000) NOT NULL,
   last_name VARCHAR2(1000) NOT NULL,
   date_of_birth DATE NOT NULL,
   phone_number VARCHAR2(1000) NOT NULL,
   CONSTRAINT employees_pk PRIMARY KEY
                              (employee_id)
);
```

…si se adauga 1000 de inregistrari in employees.

Index creat automat

Clauza **WHERE**

```
SELECT first_name, last_name
     FROM employees
     WHERE employee_id = 123
```

```
SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 3640292141

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |     1 |    33 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    33 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | EMPLOYEES_PK |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=123)

14 rows selected.
```

E mai bine unique scan sau range scan ? Un index poate avea range scan ?

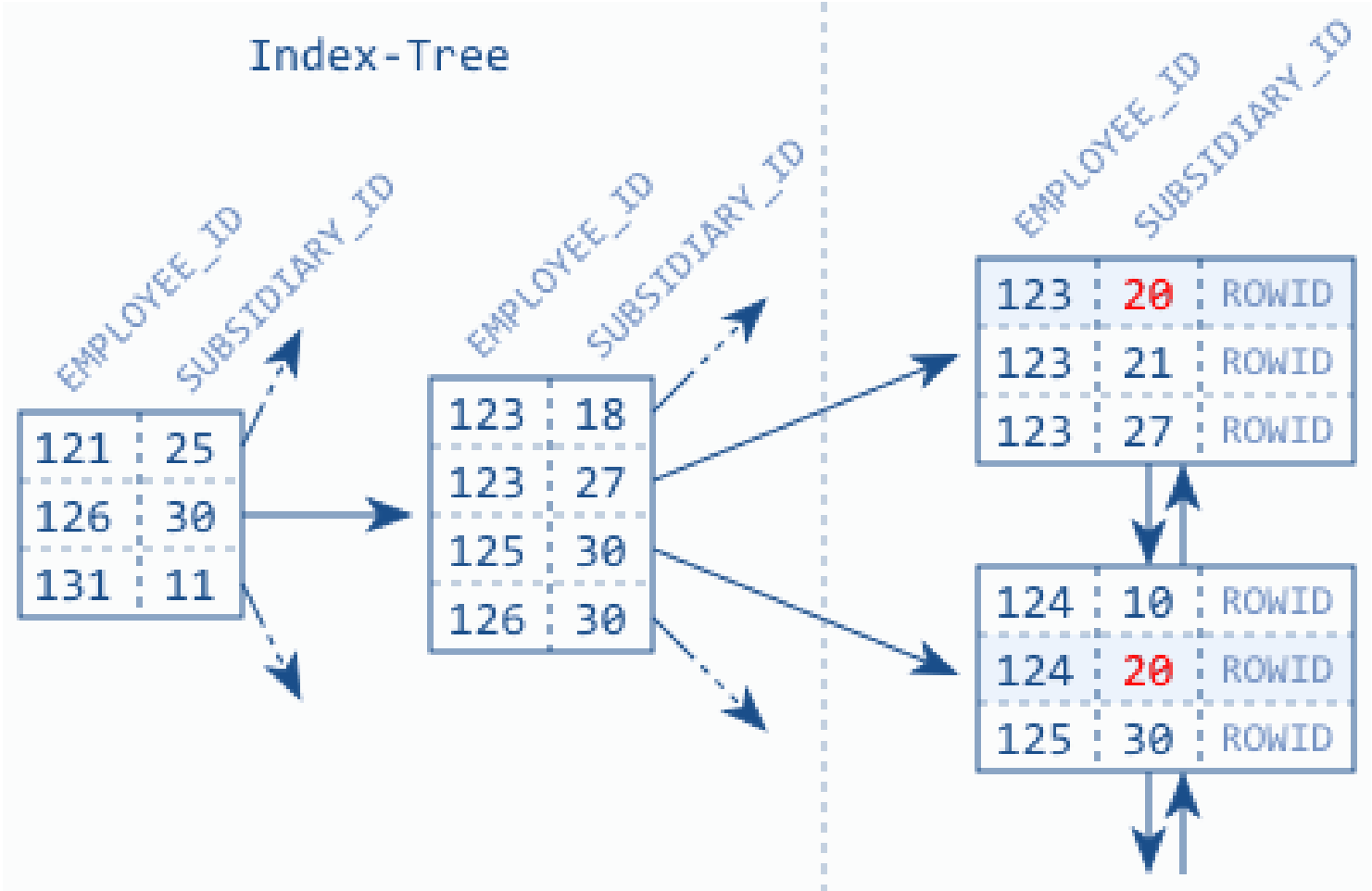
Concatenarea indecsilor

- Uneori este nevoie ca indexul sa il construim peste mai multe coloane

```
CREATE UNIQUE INDEX employee_pk ON
 employees (employee_id, subsidiary_id);
```

- Dupa ce compania a fost cumparata de o alta firma (cu 9000 de angajati), incercam iarasi sa gasim angajatul.

Concatenarea indecsilor



Clauza **WHERE**

```
SELECT first_name, last_name
    FROM employees
    WHERE employee_id = 123 AND subsidiary_id = 30
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 3640292141

------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |     1 |    36 |     2   (0)| 00:00:01
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    36 |     2   (0)| 00:00:01
|*  2 |   INDEX UNIQUE SCAN          | EMPLOYEES_PK |     1 |       |     1   (0)| 00:00:01
------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=123 AND "SUBSIDIARY_ID"=30)

14 rows selected.
```

Clauza **`WHERE`**

- Atunci cand o interogare foloseste intreaga cheie (ambele campuri constituente), cautarea este de tipul INDEX UNIQUE SCAN
- Ce se intampla daca vrem sa cautam doar dupa unul din campuri ? De exemplu dupa **`subsidiary_id`** ?

Clauza **WHERE**

```
SELECT first_name, last_name
   FROM employees
   WHERE subsidiary_id = 20
```

- Ce se va intampla ?

Clauza **WHERE**

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1445457117

------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   111 |  3996 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |   111 |  3996 |    18   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SUBSIDIARY_ID"=20)
```

Indexul nu a fost utilizat.

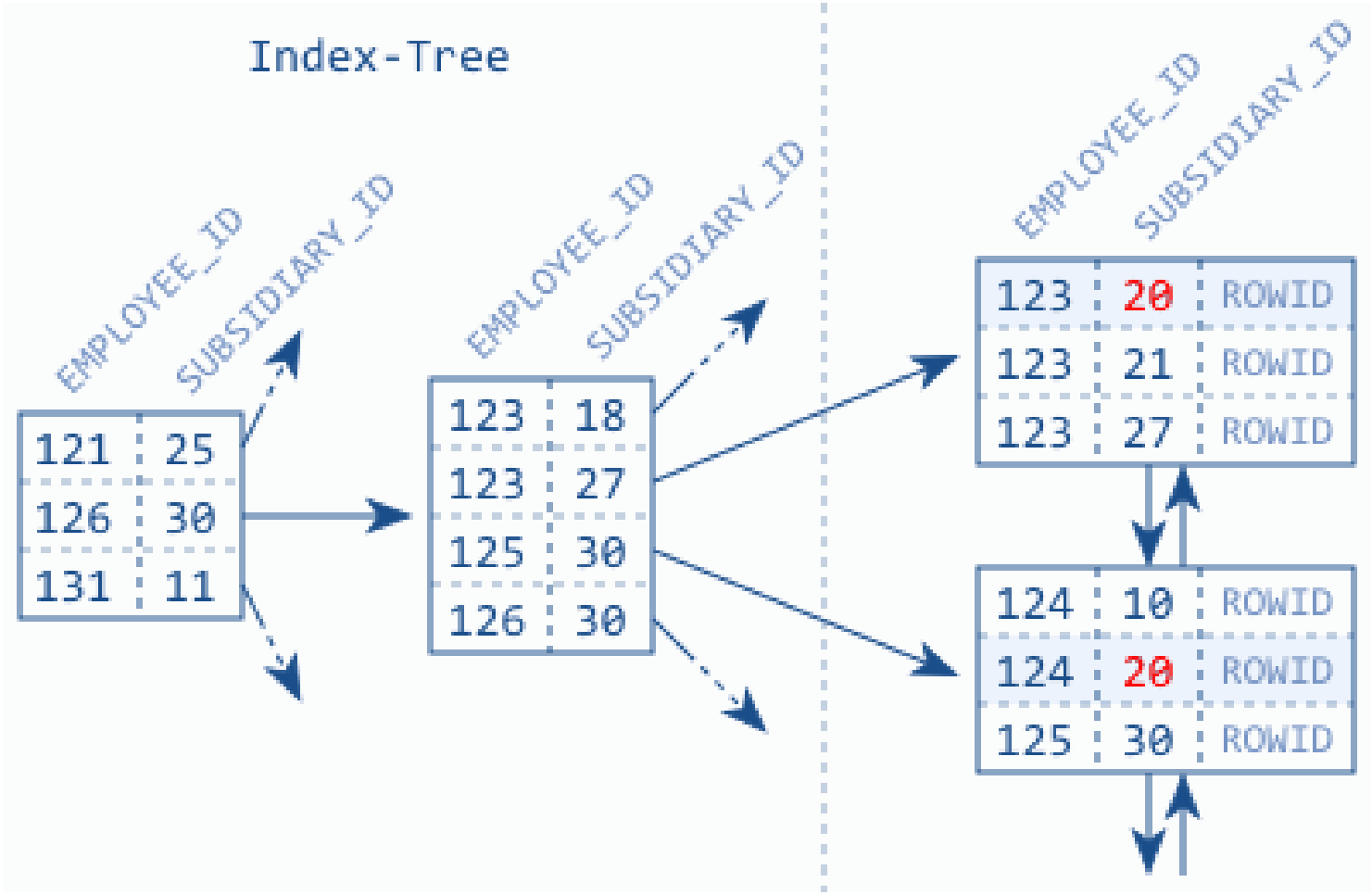
Cu cat a crescut timpul de executie ?

Operatia este rapida intr-un exemplu mic dar foarte costisitoare in caz contrar.

Clauza **WHERE**

- Uneori scanarea completa a bazei de date este mai eficienta decat accesul prin indecsi. Acest lucru este partial chiar din cauza timpului necesar cautarii indecsilor.
- O singura coloana dintr-un index concatenat nu poate fi folosita ca index (exceptie face prima coloana).

Concatenarea indecsilor – 1st column ?



Cautare dupa employee_id

Utilizarea doar a primei coloane din index este inca posibila:

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 426715952

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |     1 |    36 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    36 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMPLOYEES_PK |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=123)

14 rows selected.
```

Clauza **WHERE**

- Se observa ca valoarea 20 pentru `subsidiary_id` este distribuit aleator prin toata tabela. Din acest motiv, nu este eficient sa cautam dupa acest index.
- Cum facem ca sa cautam eficient dupa acesta?

```
DROP INDEX EMPLOYEES_PK;
CREATE UNIQUE INDEX EMPLOYEES_PK
 ON EMPLOYEES (SUBSIDIARY_ID, EMPLOYEE_ID);
```

- In continuare indexul este format din cele doua coloane.

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 426715952

-------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |   102 |  3672 |    10   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |   102 |  3672 |    10   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMPLOYEES_PK |   102 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SUBSIDIARY_ID"=20)

14 rows selected.
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1445457117

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     1 |    36 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |     1 |    36 |    18   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("EMPLOYEE_ID"=123)

13 rows selected.
```

Clauza **WHERE**

- Cel mai important lucru cand definim indecsi concatenati este sa stabilim ordinea.
- Daca vrem sa utilizam 3 campuri pentru concatenare, cautarea este eficienta pentru campul 1, pentru 1+2 si pentru 1+2+3 dar nu si pentru alte combinatii.

Clauza **WHERE**

- Atunci cand este posibil, este de preferat utilizarea unui singur index (din motive de spatiu ocupat pe disc si din motive de eficienta a operatiilor ce se efectueaza asupra bazei de date).
- Pentru a face un index compus eficient trebuie tinut cont si care din campuri ar putea fi interogate independent – acest lucru este stiut de obicei **doar de catre programator**.

Indecsi “inceti”

- Schimbarea indecsilor poate afecta intreaga baza de date ! (operatiile pe aceasta pot deveni mai greoaie din cauza ca managementul lor este facut diferit)
- Indexul construit anterior este folosit pentru toate interogarile in care este folosit **`subsidiary_id`**.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
Plan hash value: 426715952

------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |     1 |    36 |    10   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    36 |    10   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN            | EMPLOYEES_PK |   102 |       |     2   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("LAST_NAME"='WINAND')
   2 - access("SUBSIDIARY_ID"=20)
```

- Daca avem doi indecsi disjuncti si in clauza WHERE sunt folositi ambii ? Pe care dintre ei ii va considera BD? Este mereu eficient sa se tina cont de indecsi ?

Indecsi “inceti”

- ***The Query Optimizer***
- Componenta ce transforma interogarea intr-un plan de executie (aka compiling / parsing).
- Doua tipuri de opimizere:
  - Cost based optimizers (CBO) – mai multe planuri, calculeaza costul lor si ruleaza pe cel mai bun;
  - Rule-based optimizers (RBO) – foloseste un set de reguli hardcodat

CBO poate sta prea mult sa caute prin indecsi si RBO sa fie mai eficient in acest caz [1000x1000 tbl]

Indecsi “inceti”

- Cum ati executa aceasta interogare daca ati fi QO (indexul a ramas construit peste subsidiary_id, employee_id)?

```
SELECT first_name, last_name, subsidiary_id,
                                    phone_number
   FROM employees
   WHERE last_name = 'WINAND' AND
         subsidiary_id = 30
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------
Plan hash value: 1445457117

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     1 |    24 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |     1 |    24 |    18   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("LAST_NAME"='WINAND' AND "SUBSIDIARY_ID"=30)

13 rows selected.
```

- Se observa ca nu este utilizat indexul, QO preferand sa parcurga toata tabela si sa filtreze randurile corespunzatoare [si era doar un singur rand cautat].
- Un exemplu chiar mai surprinzator:

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1445457117

------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1000 | 36000 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | EMPLOYEES |  1000 | 36000 |    18   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SUBSIDIARY_ID"=30)
```

Si fara a modifica in nici un fel tabelul sau indecsii
(ci doar valoarea cautata:20 pentru subsidiary_id):

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 426715952

------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |   102 |  3672 |    10   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES    |   102 |  3672 |    10   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMPLOYEES_PK |   102 |       |     2   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SUBSIDIARY_ID"=20)
```

De ce se intampla acest lucru ?

Statistici

- CBO utilizeaza statistici despre BD (de ex. privind: tabelele, coloanele, indecsii). De exemplu, pentru o **tabela** poate memora:
  - valoarea maxima/minima,
  - numarul de valori distincte,
  - numarul de campuri NULL,
  - distributia datelor (histograma),
  - dimensiunea tabelei (nr randuri/blocuri).

Statistici

- CBO utilizeaza statistici despre BD (de ex. privind: tabelele, coloanele, indecsii). De exemplu, pentru un **index** poate memora:
  - adancimea B-tree-ului,
  - numarul de frunze,
  - numarul de chei distincte din index,
  - factor de "*clustering*".

- **Utilizarea indecsilor nu e mereu solutia cea mai potrivita.**

Indecsi bazati pe functii

Functii

- Sa presupunem ca dorim sa facem o cautare dupa **`last_name`**.

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1445457117

------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   100 |  3600 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | EMPLOYEES |   100 |  3600 |    18   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------

   1 - filter("LAST_NAME"='WINAND')

13 rows selected.
```

Functii

- Evident, aceasta cautare va fi mai rapida daca:

```
CREATE INDEX emp_name ON employees (last_name);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1035187335

------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)|
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |   100 |  3600 |    17   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |   100 |  3600 |    17   (0)|
|*  2 |   INDEX RANGE SCAN           | EMP_NAME  |    40 |       |     1   (0)|
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("LAST_NAME"='WINAND')

14 rows selected.
```

Functii

- Ce se intampla daca vreau *ignorecase*?
- Pentru o astfel de cautare, desi avem un index construit peste coloana cu `last_name`, acesta va fi ignorat [de ce ? – exemplu]

  [poate utilizarea unui alt *collation* ?!]*
- Pentru ca BD nu cunoaste rezultatul apelului unei functii a-priori, functia va trebui apelata pentru fiecare linie in parte.

*SQL Server sau MySQL nu fac distinctie intre cases cand sorteaza.

Functii

```
SELECT * FROM employees
    WHERE UPPER(last_name) = UPPER('winand');
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1445457117

------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   100 |  3600 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |   100 |  3600 |    18   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(UPPER("LAST_NAME")='WINAND')

13 rows selected.
```

Functii

- Cum vede BD interogarea ?

```
SELECT * FROM employees
    WHERE BLACKBOX(...) = 'WINAND';
```

- Se observa totusi ca partea dreapta a expresiei este evaluata o singura data. In fapt filtrul a fost facut pentru

  UPPER("last_name")='WINAND'

Functii

- Indexul va fi reconstruit peste `UPPER(last_name)`

```
drop index emp_name;
```

```
CREATE INDEX emp_up_name
  ON employees (UPPER(last_name));
```

Functii - *function-based index (***FBI***)*

```
SELECT * FROM employees
    WHERE UPPER(last_name) = UPPER('winand');
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 4246105296

------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |   100 |  3600 |    17   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |   100 |  3600 |    17   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMP_UP_NAME |    40 |       |     1   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(UPPER("LAST_NAME")='WINAND')

14 rows selected.
```

Functii

- In loc sa puna direct valoarea campului in index, un **FBI stocheaza valoarea returnata de functie**.
- Din acest motiv functia trebuie sa returneze mereu aceeasi valoare: nu sunt permise decat functii **deterministe**.
- A nu se construi FBI cu functii ce returneaza valori aleatoare sau pentru cele care utilizeaza data sistemului pentru a calcula ceva. [?!]

Functii

- Nu exista cuvinte rezervate sau optimizari pentru FBI (altele decat cele deja explicate).
- Uneori instrumentele pentru *Object relation mapping* (ORM tools) injecteaza din prima o functie de conversie a tipului literilor (upper / lower). De ex. Hibernate converteste totul in lower.
- Puteti construi proceduri stocate deterministe ca sa fie folosite in FBI. getAge ?!?!

Functii – nu indexati TOT

- De ce ? (nu are sens sa fac un index pt. *lower*)

- Incercati sa unificati caile de acces ce ar putea fi utilizate pentru mai multe interogari.
- E mai bine sa puneti indecsii peste datele originale decat daca aplicati functii peste acestea.

Parametri dinamici

Parametri dinamici (*bind parameters*, *bind variables*)

- Sunt metode alternative de a trimite informatii catre baza de date.
- In locul scrierii informatiilor direct in interogare, se folosesc constructii de tipul ? si :name (sau @name) iar datele adevarate sunt transmise din apelul API
- E “ok” sa punem valorile direct in interogare dar abordarea parametrilor dinamici are unele avantaje:

Parametri dinamici (bind parameters, bind variables)

- Avantajele folosirii parametrilor dinamici:
  - Securitate [impiedica SQL injection]
  - Performanta [obliga QO sa foloseasca acelasi plan de executie]

Parametri dinamici (bind parameters, bind variables)

- Securitate: impiedica SQL injection*

```
statement = "SELECT * FROM users
  WHERE name ='" + userName + "';“
```

Daca `userName` e modificat in `' or '1'='1`

Daca `userName` e modificat in: `a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't`

* http://en.wikipedia.org/wiki/SQL_injection

Parametri dinamici (bind parameters, bind variables)

- Avantajele folosirii parametrilor dinamici:
  - Securitate
  - Performanta
- Performanta: Baze de date (Oracle, SQL Server) pot salva (in cache) executii ale planurilor pe care le-au considera eficiente dar **DOAR** daca interogarile sunt **EXACT** la fel. Trimitand valori diferite (nedinamic), sunt formulate interogari diferite.

Si, reamintesc….

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1445457117

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1000 | 36000 |    18   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | EMPLOYEES |  1000 | 36000 |    18   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SUBSIDIARY_ID"=30)
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 426715952

-------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |   102 |  3672 |    10   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES   |   102 |  3672 |    10   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMPLOYEES_PK|   102 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SUBSIDIARY_ID"=20)
```

Parametri dinamici (bind parameters, bind variables)

- Utilizand parametri dinamici, cele doua vor fi procesate dupa acelasi plan. E mai bine ?

- Neavand efectiv valorile, se va executa planul care este considerat mai eficient daca valorile date pentru `subsidiary_id` ar fi distribuite uniform. [atentie, nu valorile din tabela ci cele din interogare !]

Parametri dinamici (bind parameters, bind variables)

- Query optimizer este ca un compilator:
  - daca ii sunt trecute valori ca si constante, se foloseste de ele in acest mod;
  - daca valorile sunt dinamice, le vede ca variabile neinitializate si le foloseste ca atare.
- Atunci de ce ar functiona mai bine cand nu sunt stiute valorile dinainte ?

Parametri dinamici (bind parameters, bind variables)

- Atunci cand este trimisa valoarea, *The query optimizer* va construi **mai multe** planuri, va stabili care este cel mai bun dupa care il va executa. In timpul asta, s-ar putea ca un plan (prestabilit), desi mai putin eficient, sa fi executat deja interogarea.
- Utilizarea parametrilor fixati e ca si cum ai compila programul de fiecare data.

Parametri dinamici (bind parameters, bind variables)

- Cine “bindeaza” variabilele poate face eficienta interogarea (programatorul): se vor folosi parametri dinamici pentru toate variabilele MAI PUTIN pentru cele pentru care se doreste sa influenteze planului de executie.

- *In all reality, there are only a few cases in which the actual values affect the execution plan. You should therefore use bind parameters if in doubt—just to prevent SQL injections.*

Parametri dinamici (bind parameters, bind variables) – exemplu Java:

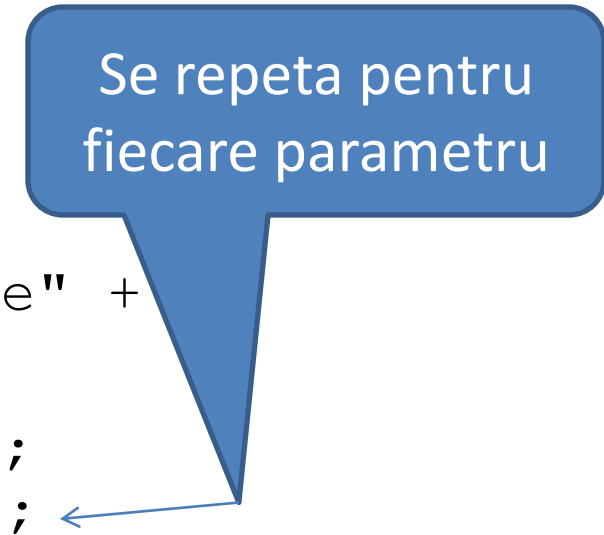
Fara bind parameters:

```
int subsidiary_id = 20;
Statement command =
  connection.createStatement(
        "select first_name, last_name" +
        " from employees" +
        " where subsidiary_id = " +
     subsidiary_id );
```

Parametri dinamici (bind parameters, bind variables) – exemplu Java:

Cu bind parameters:

```
int subsidiary_id = 20;
PreparedStatement command =
    connection.prepareStatement(
        "select first_name, last_name" +
        " from employees" +
        " where subsidiary_id = ?" );
  command.setInt(1, subsidiary_id);
  int rowsAffected =
  preparedStatement.executeUpdate();
```



http://use-the-index-luke.com/sql/where-clause/bind-parameters - C#, PHP, Perl, Java, Ruby

Parametri dinamici (bind parameters, bind variables)- Ruby:

Fara parametri dinamici:

```
dbh.execute("select first_name, last_name" +
 " from employees" +
 " where subsidiary_id = #{subsidiary_id}");
```

Cu parametri dinamici:

```
dbh.prepare("select first_name, last_name" +
    " from employees" +
    " where subsidiary_id = ?");
dbh.execute(subsidiary_id);
```

Parametri dinamici (bind parameters, bind variables)

- Semnul intrebarii indica o pozitie. El va fi indentificat prin 1,2,3… (pozitia lui) atunci cand se vor trimite efectiv parametri.
- Se poate folosi “@id” (in loc de ? si de 1,2…).

Parametri dinamici (bind parameters, bind variables)

- Parametri dinamici **nu pot schimba structura interogarii** (Ruby):

```
String sql = prepare("SELECT * FROM ?
  WHERE ?");

sql.execute('employees',
                    'employee_id = '1');
```

Pentru a schimba structura interogarii: dynamic SQL.

Cautari pe intervale

Q: Daca avem doua coloane, una dintre ele cu foarte multe valori diferite si cealalta cu foarte multe valori identice. Pe care o punem prima in index ?

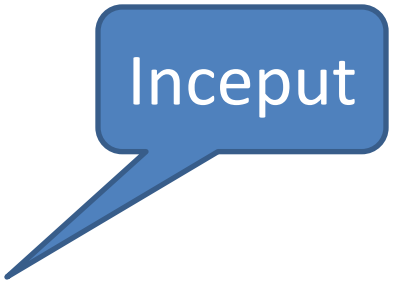
[carte de telefon:numele sunt mai diversificate decat prenumele]

Cautari pe intervale

- Sunt realizate utilizand operatorii <, > sau folosind BETWEEN.
- Cea mai mare problema a unei cautari intr-un interval este *traversarea frunzelor.*
- Ar trebui ca intervalele sa fie cat mai mici posibile. Intrebarile pe care ni le punem:
  - *unde incepe un index scan* ?
  - *unde se termina* ?

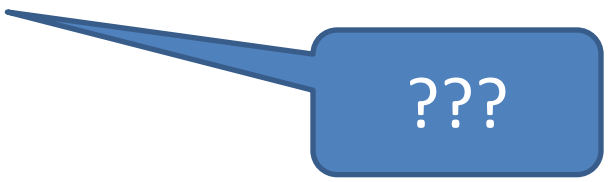
Cautari pe intervale

```
SELECT first_name, last_name,
  date_of_birth
FROM employees
WHERE
   date_of_birth >= TO_DATE(?, 'YYYY-
  MM-DD')
   AND
   date_of_birth <= TO_DATE(?, 'YYYY-
  MM-DD')
```



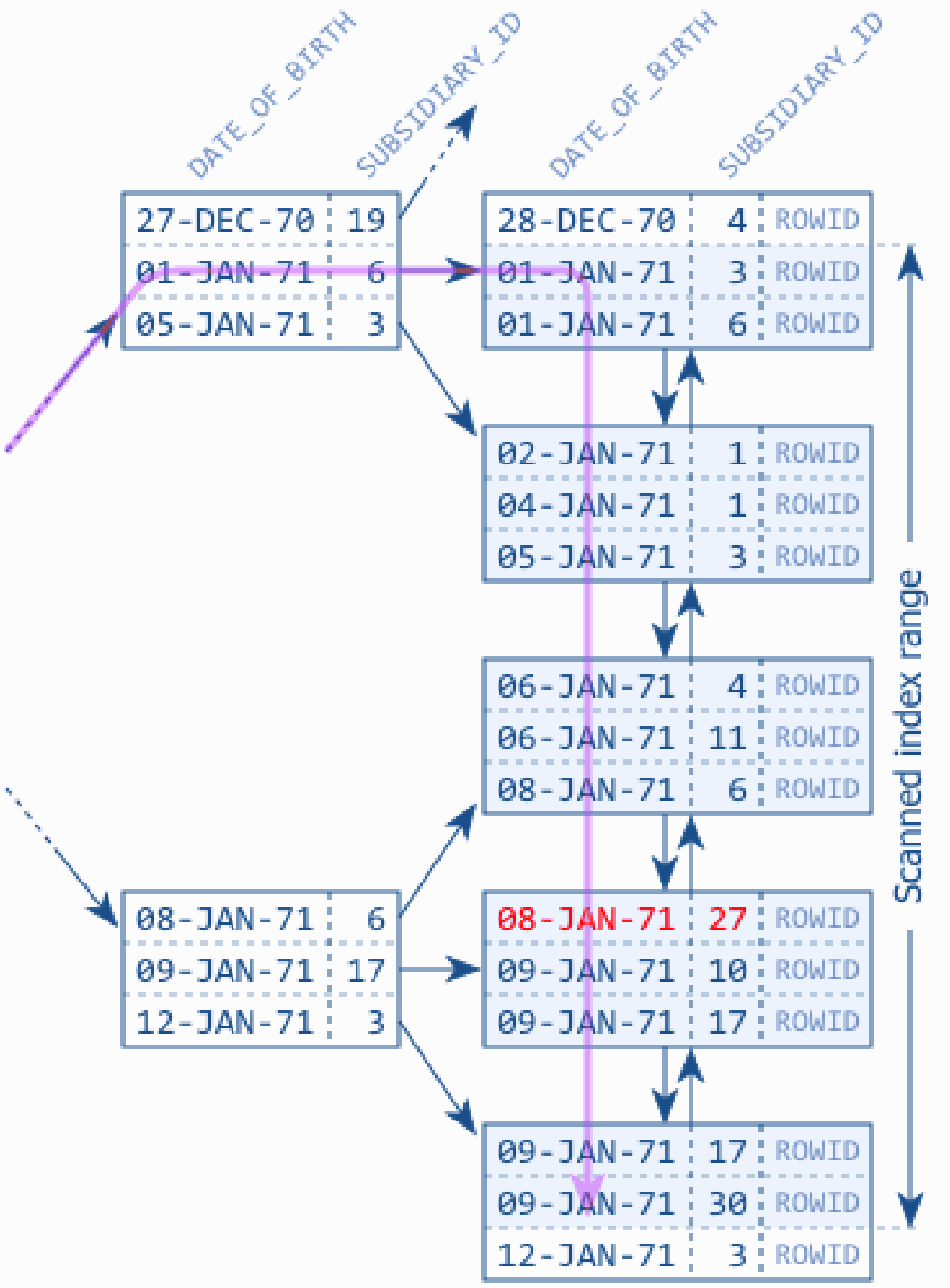
Cautari pe intervale

```
SELECT first_name, last_name,
  date_of_birth
FROM employees
WHERE
 date_of_birth >= TO_DATE(?, 'YYYY-
  MM-DD')  AND
   date_of_birth <= TO_DATE(?, 'YYYY-
  MM-DD')  AND
AND subsidiary_id = ?
```



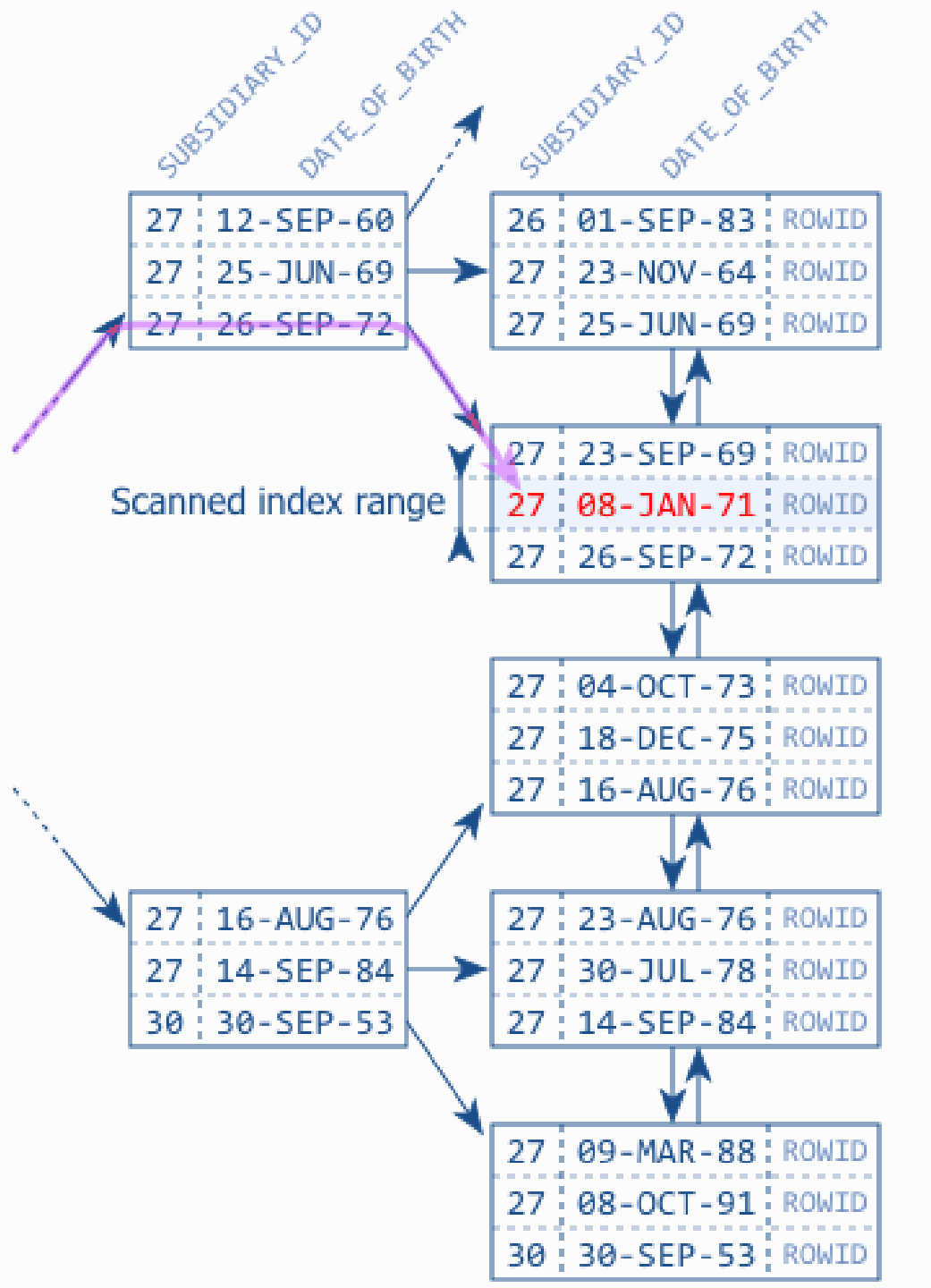
Cautari pe intervale

- Indexul ideal acopera ambele coloane.
- In ce ordine ?



1 Ianuarie 1971
9 ianuarie 1971

Sub_id=27



Sub_id=27

1 Ianuarie 1971
9 ianuarie 1971

Cautari pe intervale

Regula: indexul pentru egalitate primul
si apoi cel pentru interval !

Nu e neaparat bine ca sa punem pe prima pozitie coloana cea mai diversificata.

Cautari pe intervale

- Depinde si de ce interval cautam (pentru intervale foarte mari s-ar putea sa fie mai eficient invers).
- Nu este neaparat ca acea coloana cu valorile cele mai diferite sa fie prima in index – vezi cazul precedent in care sunt doar 30 de IDuri si 365 de zile de nastere (x ani).
- Ambele indexari faceau *match* pe 13 inregistrari.

Cautari pe intervale

```
-----------------------------------------------------------------
|Id | Operation                          | Name      | Rows | Cost |
-----------------------------------------------------------------
| 0 | SELECT STATEMENT                   |           |    1 |    4 |
|*1 |  FILTER                            |           |      |      |
| 2 |   TABLE ACCESS BY INDEX ROWID      | EMPLOYEES |    1 |    4 |
|*3 |    INDEX RANGE SCAN                | EMP_TEST  |    2 |    2 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - filter(:END_DT >= :START_DT)
3 - access(DATE_OF_BIRTH >= :START_DT
       AND DATE_OF_BIRTH <= :END_DT)
    filter(SUBSIDIARY_ID  = :SUBS_ID)
```

Dupa ce a fost gasit intervalul, datele au fost filtrate…

Deci prima coloana din index este… ?

Cautari pe intervale

- Acces – indica de unde incepe si unde se termina *rangeul* pentru cautare
- Filtrul – preia un *range* si selecteaza doar liniile care satisfac o conditie

- Daca schimbam ordinea coloanelor din index:

  (subsidiary_id, date_of_birth)

Cautari pe intervale

```
-------------------------------------------------------------------
| Id | Operation                    | Name      | Rows | Cost |
-------------------------------------------------------------------
|  0 | SELECT STATEMENT             |           |    1 |    3 |
|* 1 |  FILTER                      |           |      |      |
|  2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES |    1 |    3 |
|* 3 |    INDEX RANGE SCAN          | EMP_TEST2 |    1 |    2 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - filter(:END_DT >= :START_DT)
3 - access(SUBSIDIARY_ID  = :SUBS_ID
       AND DATE_OF_BIRTH >= :START_DT
       AND DATE_OF_BIRTH <= :END_T)
```

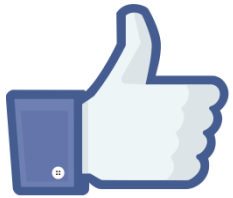
Cautari pe intervale

- Operatorul BETWEEN este echivalent cu o cautare in interval dar considerand si marginile intervalului.

```
DATE_OF_BIRTH BETWEEN '01-JAN-71'
    AND '10-JAN-71
```

Este echivalent cu:

```
DATE_OF_BIRTH >= '01-JAN-71' AND
  DATE_OF_BIRTH <= '10-JAN-71’
```



LIKE

LIKE

- Operatorul LIKE poate avea repercusiuni nedorite asupra interogarii (chiar cand sunt folositi indecsi).
- Unele interogari in care este folosit LIKE se pot baza pe indecsi, altele nu. Diferenta o face pozitia caracterului % .

LIKE

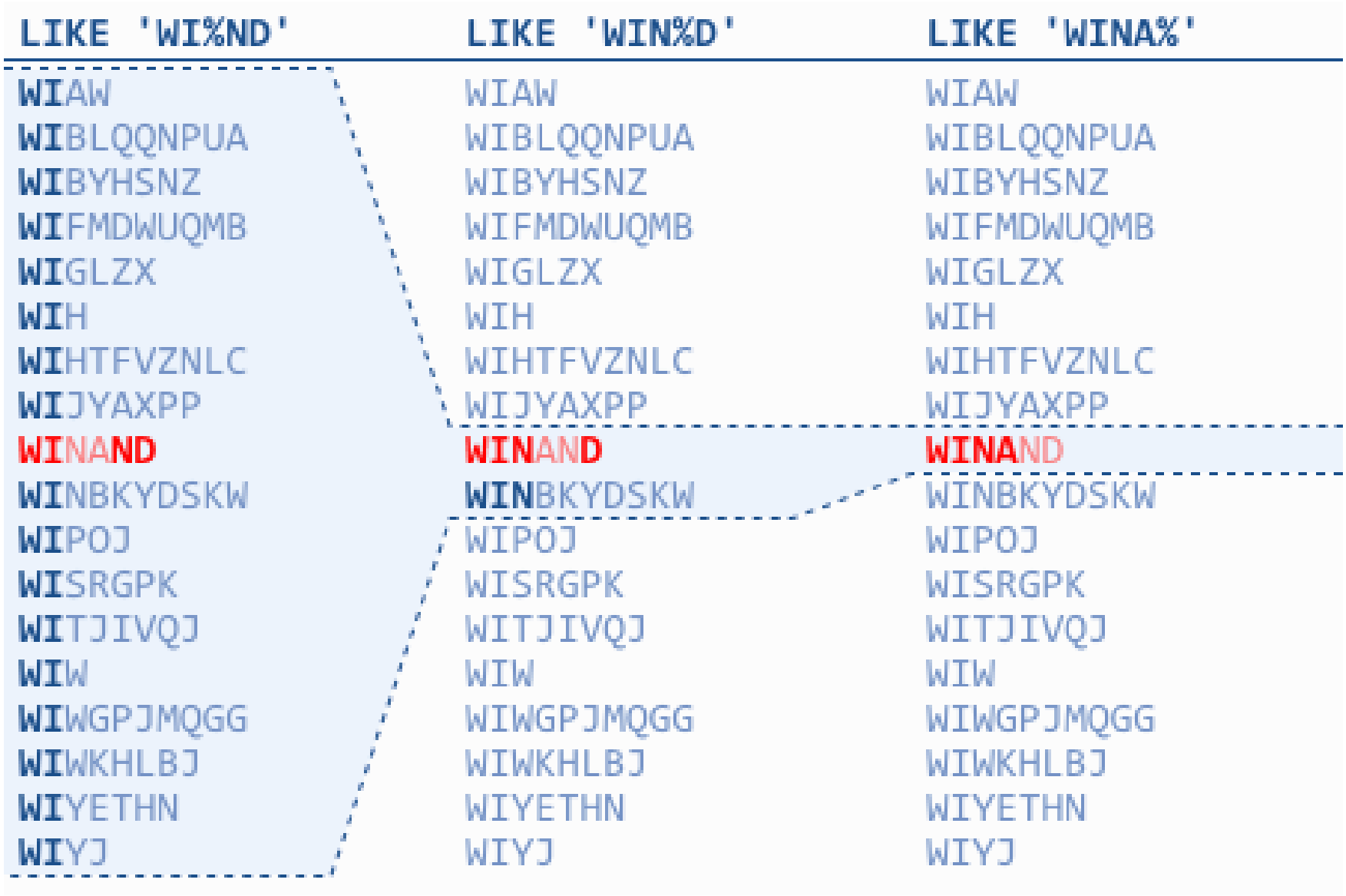
```
SELECT first_name, last_name,
  date_of_birth
FROM employees
WHERE UPPER(last_name) LIKE 'WIN%D'
```

```
-------------------------------------------------------------------
|Id | Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 | SELECT STATEMENT             |              |    1 |    4 |
| 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    4 |
|*2 |   INDEX RANGE SCAN           | EMP_UP_NAME  |    1 |    2 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(UPPER("LAST_NAME") LIKE 'WIN%D')
       filter(UPPER("LAST_NAME") LIKE 'WIN%D')
```

LIKE

- Doar primele caractere dinainte de % pot fi utilizate in cautarea bazata pe indecsi. Restul caracterelor sunt utilizate pentru a filtra rezultatele obtinute.
- Daca in exemplul anterior punem un index peste **`UPPER(last_name)`**, iata cum ar fi procesate diverse interogarile in functie in care caracterul % este asezat in diverse pozitii:

LIKE


LIKE 'WI%ND'
LIKE 'WIN%D'
LIKE 'WINA%'
WIAW
WIBLQQNPUA
WIBYHSNZ
WIFMDWUQMB
WIGLZX
WIH
WIHTFVZNLC
WIJYAXPP
WINAND
WINBKYDSKW
WIPOJ
WISRGPK
WITJIVQJ
WIW
WIWGPJMQGG
WIWKHLBJ
WIYETHN
WIYJ

LIKE

- Ce se intampla daca LIKE-ul este de forma **`LIKE '%WI%D'`** ?

LIKE

- A se evita expresii care incep cu %.
- In teorie, %, influenteaza felul in care este cautata expresia. In practica, daca sunt utilizati parametri dinamici, nu se stie cum *Querry optimizer* va considera ca este mai bine sa procedeze: ca si cum interogarea ar incepe cu % sau ca si cum ar incepe fara?

LIKE

- Daca avem de cautat un **cuvant intr-un text**, nu conteaza daca acel cuvant este trimis ca parametru dinamic sau hardcodat in interogare. Cautarea va fi oricum de tipul %cuvant% . Totusi folosind parametri dinamici macar evitam SQL injection.

LIKE

- Pentru a “optimiza” cautarile cu clauza LIKE, se pot utiliza in mod intentionat alt camp indexat (daca se stie ca intervalul ce va fi returnat de index va contine oricum textul ce contine parametrul din like).

Q: Cum ati putea indexa totusi pentru a optimiza o cautare care sa aiba ca si clauza:

```
LIKE '%WINAND'
```

Contopirea indecsilor

Indecsi de tip *Bitmap*

Contopirea indecsilor

- Este mai bine sa se utilizeze cate un index pentru fiecare coloana sau e mai bine sa fie utilizati indecsi pe mai multe coloane ?
- Sa studiem urmatoarea interogare:

```
SELECT first_name, last_name, date_of_birth
  FROM employees
  WHERE UPPER(last_name) < ?
        AND date_of_birth < ?
```

Contopirea indecsilor

- Indiferent de cum ar fi intoarsa problema, nu se poate construi un index care sa duca atat numele cat si data de nastere intr-un interval compact (cu inregistrari consecutive) [decat daca nu cumva toate mamele si-au botezat copii nascuti in Iulie drept “Iulian” ☺].
- Indexul peste doua coloane nu ne ajuta extraordinar. Totusi, daca il construim, punem coloana cu valorile cele mai diferite pe prima pozitie.

De ce ?

Contopirea indecsilor

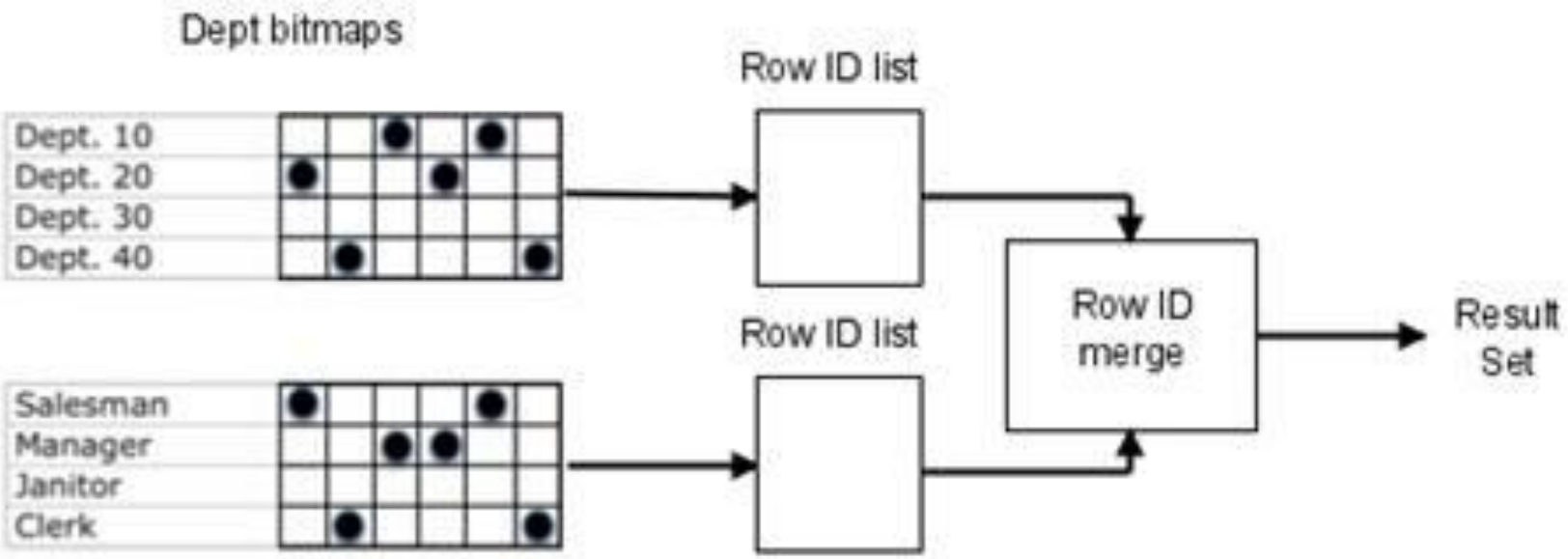
- O a doua posibilitate este utilizarea de indecsi diferiti pentru fiecare coloana si lasat QO sa decida cum sa ii foloseasca [deja s-ar putea sa ia mai mult timp pentru ca poate crea de doua ori mai multe planuri].

- Pont: o cautare dupa un singur index este mai rapida decat daca sunt doi indecsi implicati.

Contopirea indecsilor

- *Data warehouse* (DW) este cel care are grija de toate interogarile ad-hoc.
- Din cauza ca nu poate folosi un index clasic, foloseste un tip special de index: *bitmap* index (felul cum sunt indexate patratele de pe o tabla de sah).
- Merge oarecum mai bine decat fara (dar nu mai bine ca un index folosit eficient).

Bitmap Index



http://www.dba-oracle.com/oracle_tips_bitmapped_indexes.htm

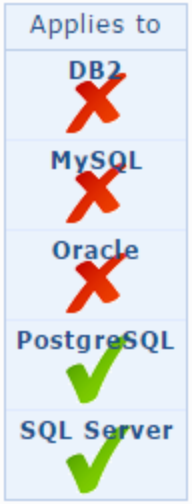
[nu este din luke…]

Contopirea indecsilor

- Dezavantajul folosirii indecsilor de tip bitmap:
  - timpul ridicol de mare pentru operatii de insert/delete/update.
  - nu permite scrieri concurente (desi in DW pot fi executate serial)
  - In aplicatii online, indecsii de tip bitmap sunt inutili
- Uneori arborii de acces (B-trees) sunt convertiti (temporar) in bitmaps de catre BD pentru a executa interogarea (nu sunt stocati) – solutie disperata a QO ce foloseste CPU+RAM.

Indecsi Partiali

Indexarea NULL



- Sa analizam interogarea:

```
SELECT message
FROM messages
WHERE processed = 'N'
      AND receiver = ?
```

- Preia toate mailurile nevizualizate (de exemlu). Cum ati indexa ? [ambele sunt cu =]

- Am putea crea un index de forma:

```
CREATE INDEX messages_todo ON
  messages (receiver, processed)
```

- Se observa ca **`processed`** imparte tabela in doua categorii: mesaje procesate si mesaje neprocesate.

Indecsi partiali

- **Unele** BD permit indexarea partiala. Asta inseamna ca indexul nu va fi creat decat peste anumite linii din tabel.

```
CREATE INDEX messages_todo
   ON messages (receiver)
   WHERE processed = 'N'
```

Atentie: nu merge in Oracle…

Indecsi partiali

- Ce se intampla la executia codului:

```
SELECT message
    FROM messages
    WHERE processed = 'N';
```

Indecsi partiali

- Indexul nou construit este redus si pe verticala (pentru ca are mai putine linii) dar si pe orizontala (nu mai trebuie sa aiba grija de coloana “processed”).
- Se poate intampla ca dimensiunea sa fie constanta (de exemplu nu am mereu ~500 de mailuri necitite) chiar daca numarul liniilor din BD creste.

NULL in Oracle

- Ce este NULL in Oracle ?
- In primul rand trebuie folosit “IS NULL” si nu “=NULL”.
- NULL nu este mereu conform standardului (ar trebui sa insemne absenta datelor).
- Oracle trateaza un sir vid ca si NULL ?!?! (de fapt trateaza ca NULL orice nu stie sau nu intelege sau care nu exista).

```
   SELECT      '0 IS NULL???' AS "what is NULL?" FROM dual
    WHERE       0 IS NULL
UNION ALL
   SELECT     '0 is not null' FROM dual
    WHERE      0 IS NOT NULL
UNION ALL
   SELECT ''''' IS NULL???'  FROM dual
    WHERE     '' IS NULL
UNION ALL
   SELECT ''''' is not null' FROM dual
    WHERE     '' IS NOT NULL
```

```
what is NULL?
--------------
0 is not null
'' IS NULL???
```

NULL in Oracle

- Mai mult, Oracle trateaza NULL ca sir vid:

```
SELECT dummy
     , dummy || ''
     , dummy || NULL
  FROM dual

D D D
- - -
X X X
```

NULL in Oracle

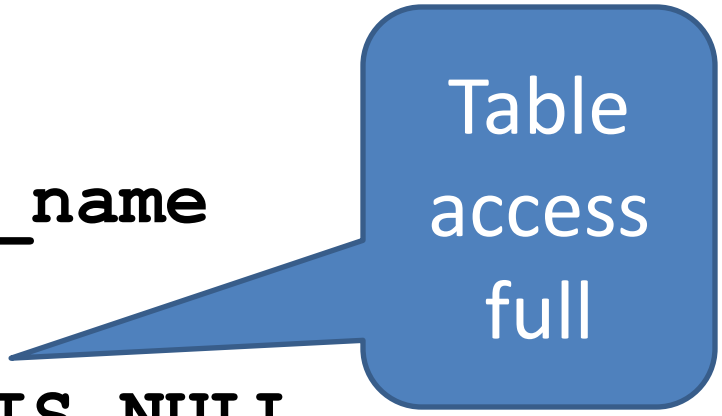
- Daca am creat un index dupa o coloana X si apoi adaugam o inregistrare care sa aiba NULL pentru X, acea inregistrare nu este indexata.

NULL in Oracle

```
INSERT INTO employees ( subsidiary_id,
  employee_id , first_name , last_name ,
  phone_number)
  VALUES ( ?, ?, ?, ?, ? );
```

- Noul rand nu va fi indexat:

```
SELECT first_name, last_name
    FROM employees
    WHERE date_of_birth IS NULL
```




```
alter table employees modify (date_of_birth null);
```

Indexarea NULL in Oracle

```
CREATE INDEX demo_null ON employees
  (subsidiary_id, date_of_birth);
```

- Si apoi:

```
SELECT first_name, last_name
    FROM employees
    WHERE subsidiary_id = ?
          AND date_of_birth IS NULL
```

Indexarea NULL in Oracle

```
-----------------------------------------------------------------
| Id | Operation                    | Name        | Rows | Cost |
-----------------------------------------------------------------
|  0 | SELECT STATEMENT             |             |    1 |    2 |
|  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |    1 |    2 |
|* 2 |   INDEX RANGE SCAN           | DEMO_NULL |    1 |    1 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("SUBSIDIARY_ID"=TO_NUMBER(?)
         AND "DATE_OF_BIRTH" IS NULL)
```

- Ambele predicate sunt utilizate !

Indexarea NULL in Oracle

- Atunci cand indexam dupa un camp ce s-ar putea sa fie NULL, pentru a ne asigura ca si aceste randuri sunt indexate, trebuie adaugat un camp care sa fie NOT NULL ! (poate fi adaugat si o constanta – de exemplu ‘1’):

```
DROP INDEX emp_dob;
CREATE INDEX emp_dob ON employees
  (date_of_birth, '1');
```

NOT NULL

```
DROP INDEX emp_dob;
CREATE INDEX emp_dob_name
          ON employees (date_of_birth, last_name);


SELECT *
  FROM employees
 WHERE date_of_birth IS NULL


-------------------------------------------------------------------------
|Id |Operation                     | Name          | Rows | Cost |
-------------------------------------------------------------------------
| 0 |SELECT STATEMENT              |               |    1 |    3 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |    1 |    3 |
|*2 |  INDEX RANGE SCAN            | EMP_DOB_NAME  |    1 |    2 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DATE_OF_BIRTH" IS NULL)
```

```
ALTER TABLE employees MODIFY last_name NULL;


SELECT *
  FROM employees
 WHERE date_of_birth IS NULL


-------------------------------------------------------
| Id | Operation          | Name      | Rows | Cost |
-------------------------------------------------------
|  0 | SELECT STATEMENT   |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |    1 |  477 |
-------------------------------------------------------
```

- Fara NOT NULL pus pe last_name (care e folosit in index), indexul este inutilizabil.
- Se gandeste ca poate exista cazul cand ambele campuri sunt nule si acel caz nu e bagat in index.

Indexarea NULL in Oracle

- O functie creata de utilizator este considerata ca fiind NULL (indiferent daca este sau nu).

- Exista anumite functii din Oracle care sunt recunoscute ca intorc NULL atunci cand datele de intrare sunt NULL (de exemplu functia upper).

```
CREATE OR REPLACE FUNCTION blackbox(id IN NUMBER) RETURN NUMBER
DETERMINISTIC
IS BEGIN
   RETURN id;
END;

DROP INDEX emp_dob_name;
CREATE INDEX emp_dob_bb
    ON employees (date_of_birth, blackbox(employee_id));


SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;

-----------------------------------------------------
| Id | Operation          | Name      | Rows | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT   |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |    1 |  477 |
-----------------------------------------------------
```

In opinia lui, ambele pot fi NULL.

Desi id este NOT NULL

Ii spunem clar ca nu ne intereseaza unde functia da NULL.

```
SELECT *
  FROM employees
 WHERE date_of_birth IS NULL
   AND blackbox(employee_id) IS NOT NULL;

-----------------------------------------------------------------
|Id |Operation                    | Name        | Rows | Cost |
-----------------------------------------------------------------
| 0 |SELECT STATEMENT             |             |    1 |    3 |
| 1 | TABLE ACCESS BY INDEX ROWID | EMPLOYEES   |    1 |    3 |
|*2 |  INDEX RANGE SCAN           | EMP_DOB_BB  |    1 |    2 |
-----------------------------------------------------------------
```

```
ALTER TABLE employees ADD bb_expression
      GENERATED ALWAYS AS (blackbox(employee_id)) NOT NULL;

DROP   INDEX emp_dob_bb;
CREATE INDEX emp_dob_bb
    ON employees (date_of_birth, bb_expression);


SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;
```

Si folosim coloana in index

Sau ii spunem ca acest camp este mereu NOT NULL.

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    3 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    3 |
|*2 |  INDEX RANGE SCAN           | EMP_DOB_BB   |    1 |    2 |
-------------------------------------------------------------------
```

```
DROP INDEX emp_dob_bb;

CREATE INDEX emp_dob_upname
    ON employees (date_of_birth, upper(last_name));

SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;
```

Desi ar putea fi null, stie ca atunci cand primeste un nenull va intoarce nenull.

```
-----------------------------------------------------------
|Id |Operation                    | Name            | Cost |
-----------------------------------------------------------
| 0 |SELECT STATEMENT             |                 |    3 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES       |    3 |
|*2 |  INDEX RANGE SCAN           | EMP_DOB_UPNAME  |    2 |
-----------------------------------------------------------
```

```
ALTER TABLE employees MODIFY last_name NULL;

SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;

-------------------------------------------------
| Id | Operation          | Name      | Rows | Cost |
-------------------------------------------------
|  0 | SELECT STATEMENT   |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |    1 |  477 |
-------------------------------------------------
```

Emularea indecsilor partiali in Oracle

```
CREATE INDEX messages_todo
    ON messages (receiver)
    WHERE processed = 'N'
```

- Avem nevoie de o functie care sa returneze NULL de fiecare data cand mesajul a fost procesat.

Emularea indecsilor partiali in Oracle

```
CREATE OR REPLACE FUNCTION
  pi_processed(processed CHAR,
  receiver NUMBER)
RETURN NUMBER DETERMINISTIC AS
BEGIN
      IF processed IN ('N')
            THEN RETURN receiver;
            ELSE RETURN NULL;
       END IF;
END; /
```

Pentru a putea fi utilizata in index.

```
CREATE INDEX messages_todo
          ON messages (pi_processed(processed, receiver));

SELECT message
  FROM messages
 WHERE pi_processed(processed, receiver) = ?
```

Deoarece stie ca aici va veni o valoare,QO face un singur plan (cu index). Daca ar fi fost null ar fi fost testat cu “IS NULL”.

```
-------------------------------------------------------------
|Id | Operation                    | Name          | Cost |
-------------------------------------------------------------
| 0 | SELECT STATEMENT             |               | 5330 |
| 1 |  TABLE ACCESS BY INDEX ROWID| MESSAGES      | 5330 |
|*2 |   INDEX RANGE SCAN           | MESSAGES_TODO | 5303 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("PI_PROCESSED"("PROCESSED","RECEIVER")=:X)
```

Conditii obfuscate

Metode de Ofuscare – siruri numerice

- Sunt numere memorate in coloane de tip text
- Desi nu e practic, un index poate fi folosit peste un astfel de sir de caractere:

```
SELECT ... FROM ... WHERE
  numeric_string = '42'
```

- Daca s-ar face o cautare de genul:

Metode de Ofuscare – siruri numerice

```
SELECT ... FROM ... WHERE
  numeric_string = 42
```

- Unele SGBDuri vor semnala o eroare (PostgreSQL) in timp ce altel vor face o conversia astfel:

```
SELECT ... FROM ... WHERE
  TO_NUMBER(numeric_string) = 42
```

Metode de Ofuscare – siruri numerice

- Problema este ca nu ar trebui sa convertim sirul de caractere din tabel ci mai degraba sa convertim numarul (pentru ca indexul e pe sir):

```
SELECT ... FROM ... WHERE
        numeric_string = TO_CHAR(42)
```

- De ce nu face baza de date conversia in acest mod ? Pentru ca datele din tabel ar putea fi stocate ca ‘42’ dar si ca ‘042’, ‘0042’ care sunt diferite ca si siruri de caractere dar reprezinta acelasi numar.

Metode de Ofuscare – siruri numerice

- Conversia se face din siruri in numere deoarece ‘42’ sau ‘042’ vor avea aceeasi valoare cand sunt convertite. Totusi 42 nu va putea fi vazut ca fiind atat ‘42’ cat si ‘042’ cand este convertit in sir numeric.
- Diferenta nu este numai una de performanta dar chiar una ce tine de semantica.

Metode de Ofuscare – siruri numerice

- Utilizarea sirurilor numerice intr-o tabela este problematica (de exemplu din cauza ca poate fi stocat si altceva decat un numar).
- Regula: folositi tipuri de date numerice ca sa stocati numere.

Metode de Ofuscare - Date

- Data include o componenta temporala
- Trunc(DATE) seteaza data la miezul noptii.

```
SELECT ... FROM sales WHERE
  TRUNC(sale_date) =
    TRUNC(sysdate - INTERVAL '1' DAY)
```

Nu va merge corect daca indexul este pus pe sale_date deoarece TRUNC=*blackBox*.

```
CREATE INDEX index_name ON table_name
  (TRUNC(sale_date))
```

Metode de Ofuscare - Date

- Este bine ca indecsii sa ii punem peste datele originale (si nu peste functii).
- Daca facem acest lucru putem folosi acelasi index si pentru cautari ale vanzarilor de ieri dar si pentru cautari a vanzarilor din ulrima saptamana / luna sau din luna N.

Metode de Ofuscare - Date

```
SELECT ... FROM sales WHERE
  DATE_FORMAT(sale_date, "%Y-%M") =
  DATE_FORMAT(now() , "%Y-%M')
```

- Cauta vanzarile din luna curenta. Mai rapid este:

```
SELECT ... FROM sales WHERE
  sale_date BETWEEN month_begin(?)
  AND month_end(?)
```

Metode de Ofuscare - Date

- Regula: scrieti interogarile pentru perioada ca si conditii explicite (chiar daca e vorba de o singura zi).

```
sale_date >= TRUNC(sysdate) AND
  sale_date < TRUNC(sysdate +
  INTERVAL '1' DAY)
```

Metode de Ofuscare - Date

- O alta problema apare la compararea tipurilor **date** cu **siruri de caractere**:

```
SELECT ... FROM sales WHERE
  TO_CHAR(sale_Date, 'YYYY-MM-DD') = '1970-
  01-01'
```

- Problema este (iarasi) conversia coloanei ce reprezinta data.
- Oamenii traiesc cu impresia ca parametrii dinamici trebuie sa fie numere sau caractere. In fapt ele pot fi chiar si de tipul `java.util.Date`

Metode de Ofuscare - Date

- Daca nu puteti trimite chiar un obiect de tip Date ca parametru, macar nu faceti conversia coloanei (evitand a utiliza indexul). Mai bine:


```
SELECT ... FROM sales WHERE sale_date
  = TO_DATE('1970-01-01', 'YYYY-MM-
  DD')
```

Metode de Ofuscare - Date

- Cand sale_date contine o data de tip timp, e mai bine sa utilizam intervale) :

```
SELECT ... FROM sales WHERE
    sale_date >= TO_DATE('1970-01-01',
  'YYYY-MM-DD') AND
    sale_date < TO_DATE('1970-01-01',
  'YYYY-MM-DD') + INTERVAL '1' DAY

sale_date LIKE SYSDATE
```

Metode de Ofuscare - Math

- Putem crea un index pentru ca urmatoarea interogare sa functioneze corect?

```
SELECT numeric_number FROM table_name
WHERE numeric_number - 1000 > ?
```

- Dar pentru:

```
SELECT a, b FROM table_name
WHERE 3*a + 5 = b
```

Metode de Ofuscare - Math

- In mod normal nu este bine sa punem SGBD-ul sa rezolve ecuatii.
- Pentru el, si urmatoarea interogare va face full scan:

```
SELECT numeric_number FROM table_name
WHERE numeric_number + 0 > ?
```

- Totusi am putea indexa in felul urmator:

```
CREATE INDEX math ON table_name (3*a - b)
SELECT a, b FROM table_name
WHERE 3*a - b = -5;
```

Metode de Ofuscare – "Smart logic"

```
SELECT first_name, last_name,
  subsidiary_id, employee_id FROM
  employees WHERE
( subsidiary_id = :sub_id OR :sub_id
  IS NULL ) AND
( employee_id = :emp_id OR :emp_id IS
  NULL ) AND
( UPPER(last_name) = :name OR :name
  IS NULL )
```

Metode de Ofuscare – “Smart logic”

- Cand nu se doreste utilizarea unuia dintre filtre, se trimite NULL in parametrul dinamic.
- Baza de date nu stie care dintre filtre este NULL si din acest motiv se asteapta ca toate pot fi NULL => TABLE ACCESS FULL + filtru (chiar daca exista indecsi).
- Problema este ca QO trebuie sa gaseasca planul de executie care sa acopere toate cazurile (inclusiv cand toti sunt NULL), pentru ca va folosi acelasi plan pentru interogarile cu var. dinamice.

Metode de Ofuscare – “Smart logic”

- Solutia este sa ii zicem BD ce avem nevoie si atat:

```
SELECT first_name, last_name,
  subsidiary_id, employee_id FROM
  employees
WHERE UPPER(last_name) = :name
```

- Problema apare din cauza *share execution plan* pentru parametrii dinamici.

Performanta - Volumul de date

Volumul de date

- O interogare devine mai lenta cu cat sunt mai multe date in baza de date

- Cat de mare este impactul asupra performantei daca volumul datelor se dubleaza ?
- Cum putem imbunatati ?

Volumul de date

- Interogarea analizata:

```
SELECT count(*) FROM scale_data
  WHERE section = ? AND id2 = ?
```

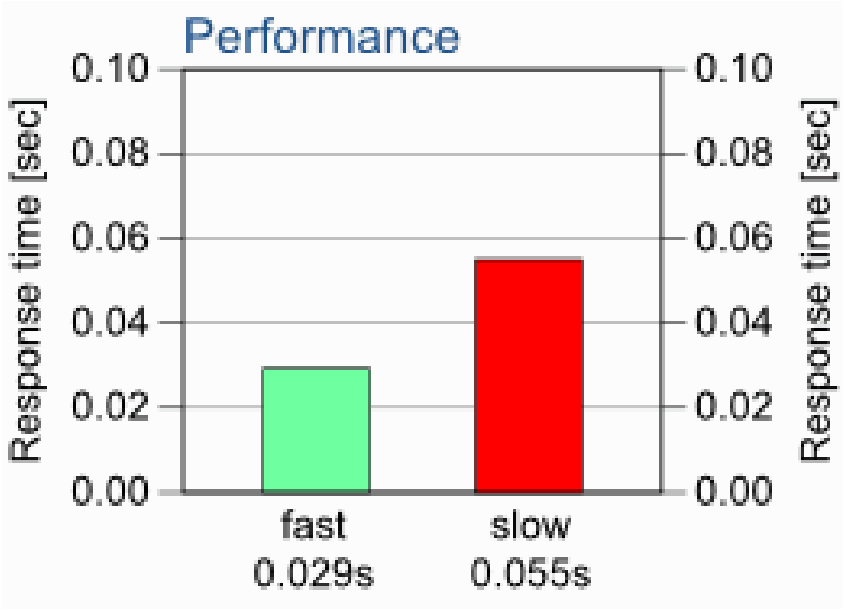
- Section are rolul de a controla volumul de date. Cu cat este mai mare section, cu atat este mai mare volumul de date returnat.
- Consideram doi indecsi: index1 si index2

Volumul de date

- Interogarea analizata:

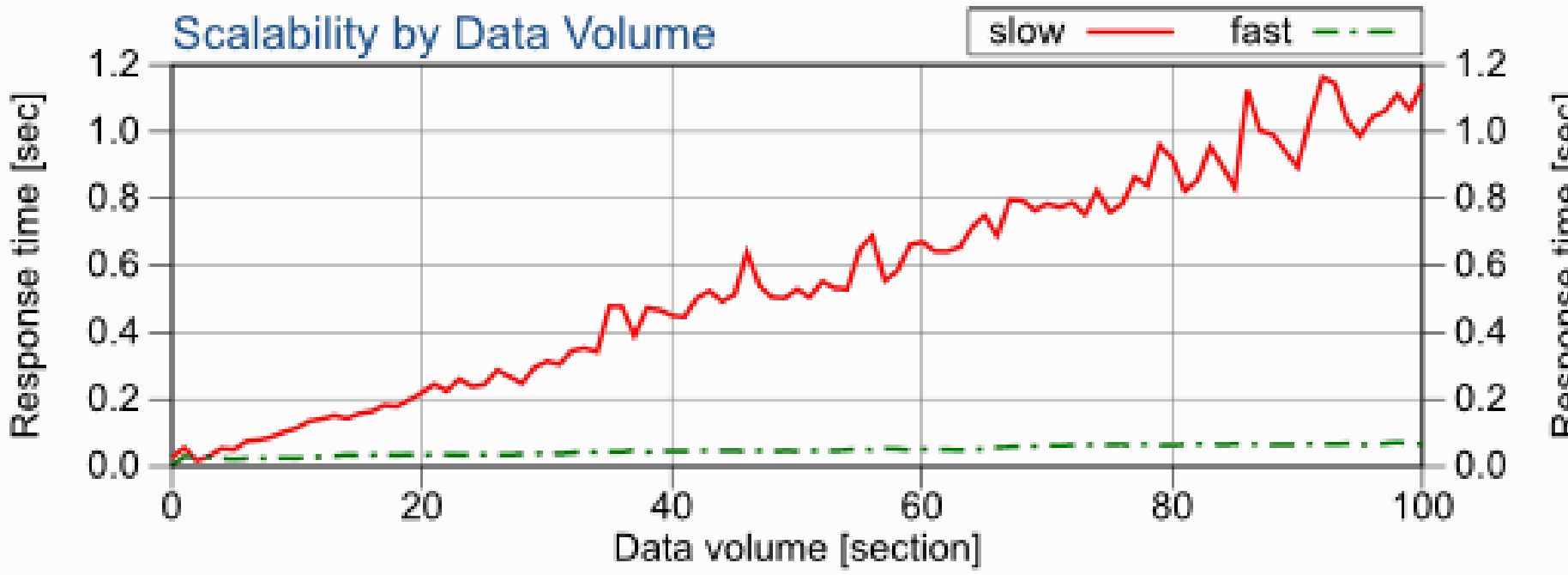
```
SELECT count(*) FROM scale_data
  WHERE section = ? AND id2 = ?
```

- *Section mic* – index1 si apoi index2



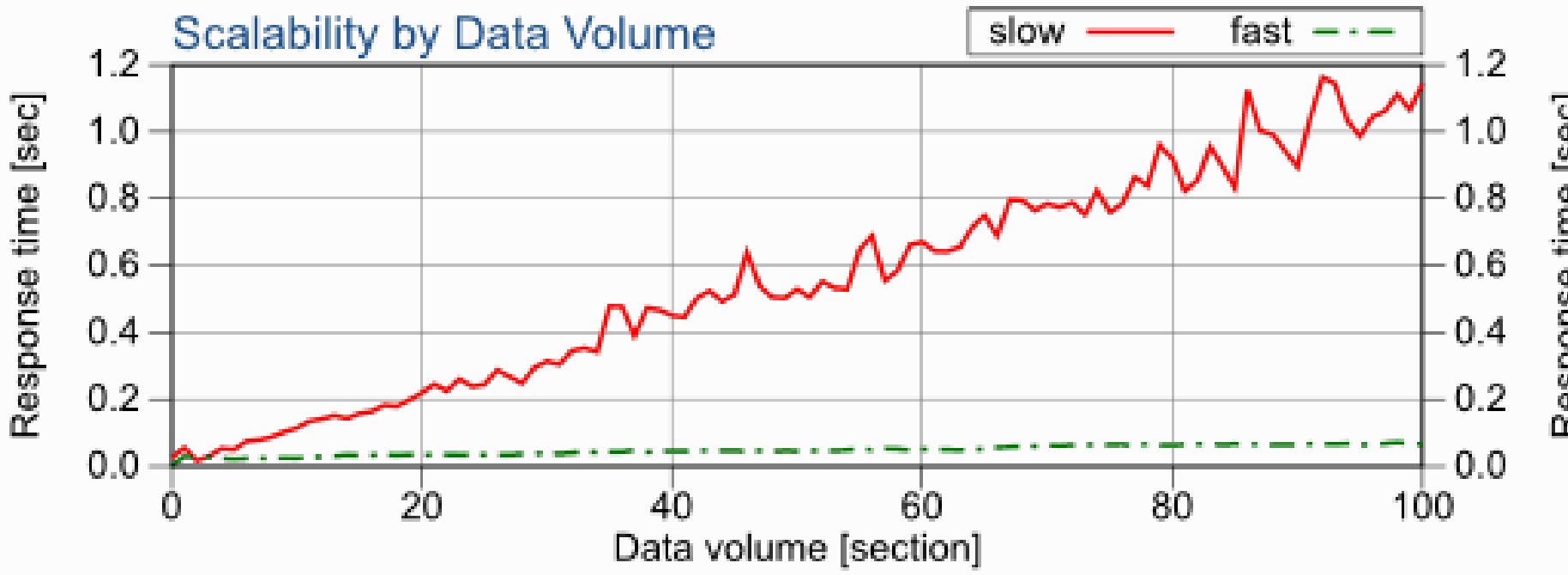
Volumul de date

- **Scalabilitatea** indica dependenta performantei in functie de factori precum **volumul de informatii.**



Volumul de date

- index1 – timp dublu fata de cel initial
- index2 – trimp x20 fata de cel initial



Volumul de date

- Raspunsul unei interogari depinde de mai multi factori. Volumul de date e unul dintre ei.
- Daca o interogare merge bine in faza de test, nu e neaparat ca ea sa functioneze bine si in productie.
- Care este motivul pentru care apare diferenta dintre index1 si index2 ?

Ambele par identice ca executie:

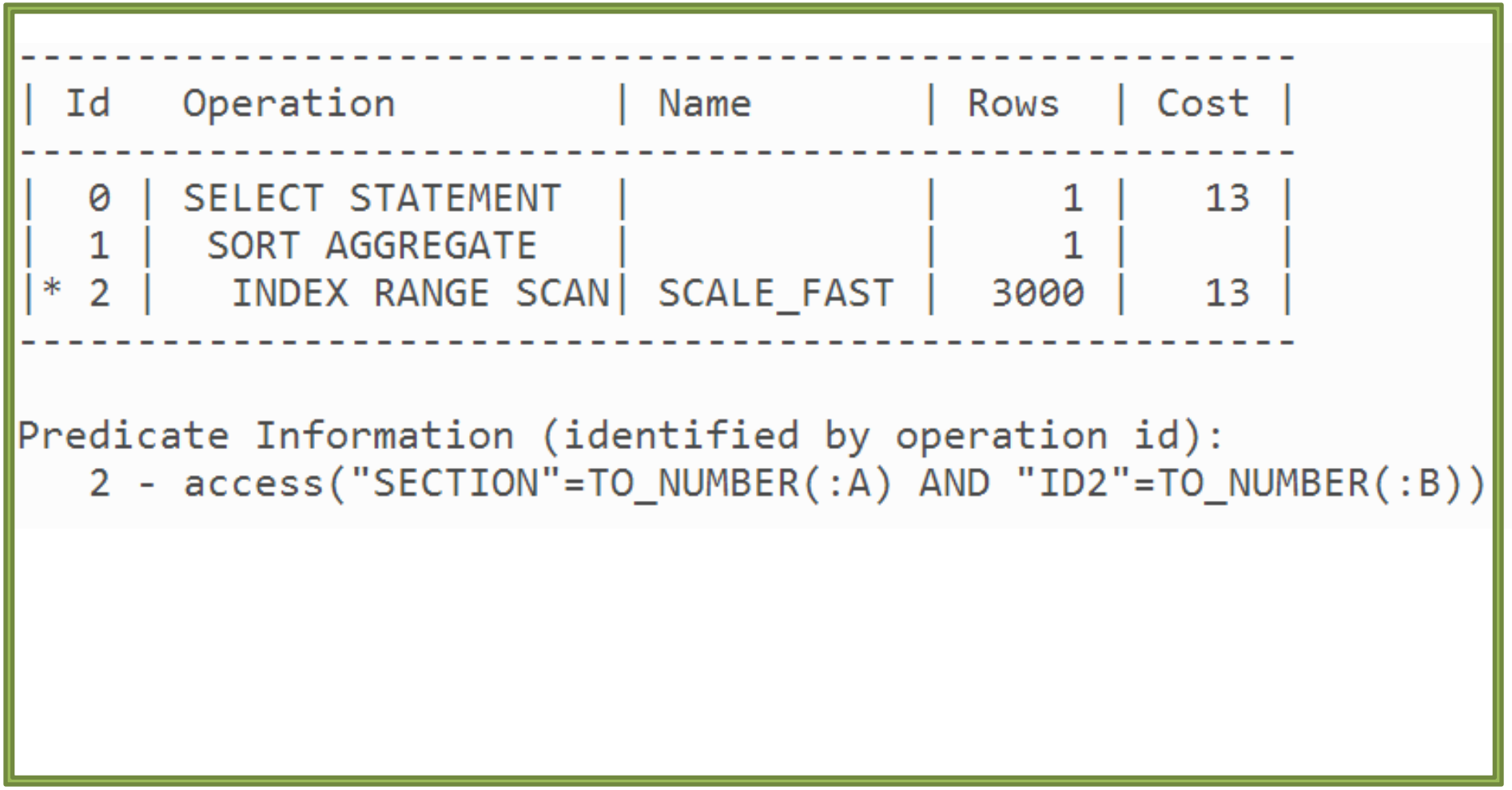
Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	**972**
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	**SCALE_SLOW**	3000	**972**

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	**13**
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	**SCALE_FAST**	3000	**13**

Volumul de date

- Ce influenteaza un index ?
  - table acces
  - scanarea unui interval mare
- Nici unul din planuri nu indica acces pe baza indexului (`TABLE ACCES BY INDEX ROW ID`)
- Unul din intervale este mai mare atunci cand e parcurs…. trebuie sa avem acces la “*predicate information*” ca sa vedem de ce:

```
-------------------------------------------------------------------
| Id | Operation          | Name        | Rows  | Cost |
-------------------------------------------------------------------
|  0 | SELECT STATEMENT   |             |     1 |  972 |
|  1 |  SORT AGGREGATE    |             |     1 |      |
|* 2 |   INDEX RANGE SCAN | SCALE_SLOW  |  3000 |  972 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
   2 - access("SECTION"=TO_NUMBER(:A))
       filter("ID2"=TO_NUMBER(:B))
```



```
-----------------------------------------------------------
| Id   Operation          | Name        | Rows   | Cost |
-----------------------------------------------------------
|  0 | SELECT STATEMENT   |             |      1 |    13 |
|  1 |  SORT AGGREGATE    |             |      1 |       |
|* 2 |   INDEX RANGE SCAN| SCALE_FAST  |   3000 |    13 |
-----------------------------------------------------------

Predicate Information (identified by operation id):
   2 - access("SECTION"=TO_NUMBER(:A) AND "ID2"=TO_NUMBER(:B))
```

Volumul de date

- Puteti spune cum a fost construit indexul avand planurile de executie ?

Volumul de date

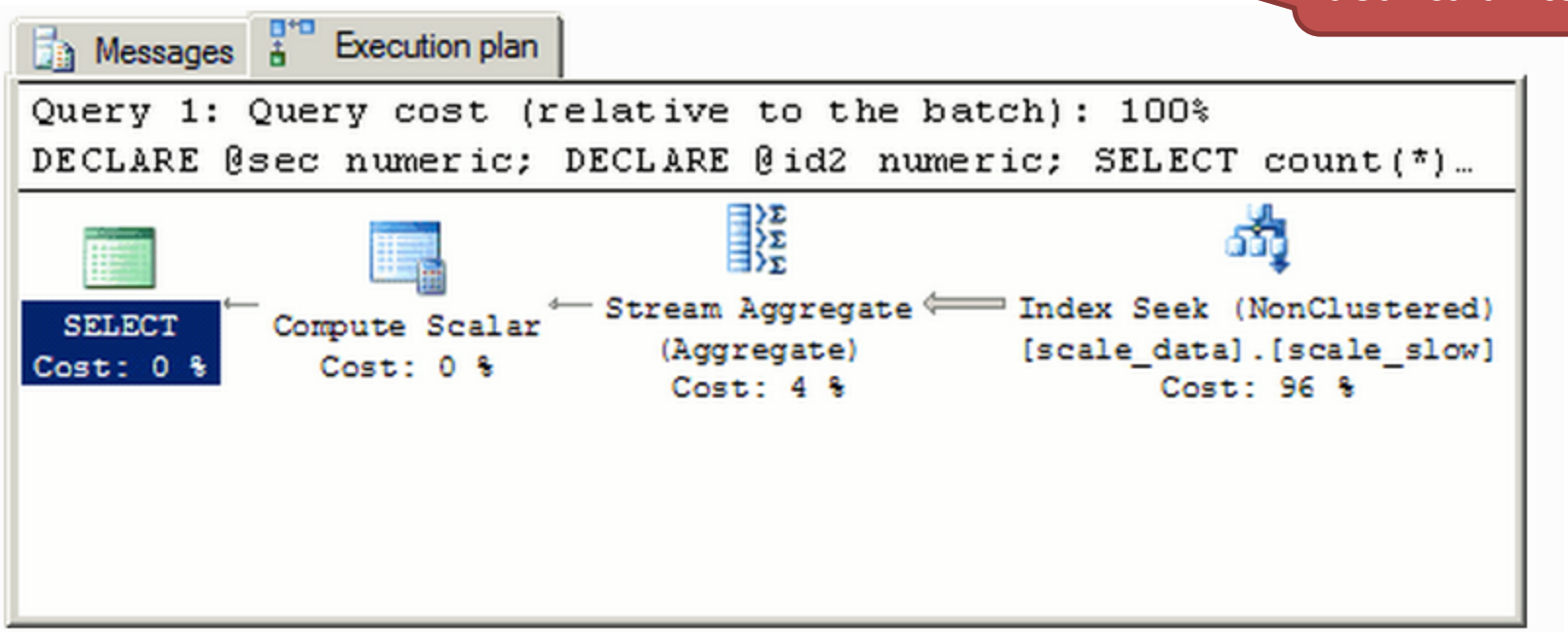
- Puteti spune cum a fost construit indexul avand execution plans ?
- `CREATE INDEX scale_slow ON scale_data (section, id1, id2);`
- `CREATE INDEX scale_fast ON scale_data (section, id2, id1);`

Campul id1 este adaugat doar pentru a pastra aceeasi dimensiune (sa nu se creada ca indexul *scale_fast* e mai rapid pentru ca are mai putine campuri in el).

Incarcarea sistemului

- Faptul ca am definit un index pe care il consideram bun pentru interogarile noastre nu il face sa fie neaparat folosit de QO.
- *SQL Server Management Studio*

Arata predicatul doar ca un tooltip



Incarcarea sistemului

- De regula, impreuna cu numarul de inregistrari, creste si numarul de accesari.
- **Numarul de accesari** este alt parametru ce intra in calculul scalabilitatii.

Incarcarea sistemului

- Daca initial era doar o singura accesare, considerand acelasi scenariu dar cu 1-25 interogari concurente, timpul de raspuns creste:

Incarcarea sistemului

- Asta inseamna ca si daca avem toata baza de date din productie si testam totul pe ea, tot sunt sanse ca in realitate, din cauza numarului mare de interogari, sa mearga mult mai greu.

- **Nota:** atentia data planului de executie este mai importanta decat *benchamarkuri* superficiale *(gen SQL Server Management Studio)*.

Incarcarea sistemului

- Ne-am putea astepta ca **hardwareul mai puternic din productie sa duca mai bine sistemul**. In fapt, in faza de development nu exista deloc latenta – ceea ce nu se intampla in productie (unde accesul poate fi intarziat din cauza retelei).
- http://blog.fatalmind.com/2009/12/22/latency-security-vs-performance/
- http://jamesgolick.com/2010/10/27/we-are-experiencing-too-much-load-lets-add-a-new-server..html

Timpi de raspuns + *throughput*

- Hardware mai performant nu este mai rapid doar poate duce mai multa incarcare.highway
- Procesoarele single-core vs procesoarele multi-core (cand e vorba de un singur task).
- Scalarea pe orizontala (adaugarea de procesoare) are acelasi efect.
- Pentru a imbunatati timpul de raspuns este necesar un arbore eficient (chiar si in NoSQL).

Timpi de raspuns

- Indexarea corecta fac cautarea intr-un B-tree in timp logaritmic.
- Sistemele bazate pe NoSQL par sa fi rezolvat problema performantei prin scalare pe orizontala [analogie cu indecsii partiali in care fiecare partitie este stocata pe o masina diferita].
- Aceasta scalabilitate este totusi limitata la operatiile de scriere intr-un model denumit "eventual consistency" [Consistency / Availability / Partition tolerance = CAP theorem] http://en.wikipedia.org/wiki/CAP_theorem

Timpi de raspuns

- Mai mult hardware de obicei nu imbunatateste sistemul.
- *Latency* al HDD [problema apare cand datele sunt culese din locatii diferite ale HDDului – de exemplu in cadrul unei operatii JOIN]. SSD?

"Facts"

- *Performance has two dimensions: response time and throughput.*
- *More hardware will typically not improve query response time.*
- *Proper indexing is the best way to improve query response time.*

Join

An SQL query walks into a bar and sees two tables.

He walks up to them and asks 'Can I join you?'

— Source: Unknown

Join

- *Join-ul* transforma datele dintr-un **model normalizat** intr-unul **denormalizat** care serveste unui anumit scop.
- Sensibil la latente ale discului (si fragmentare).

Join

- Reducerea timpilor = indexarea corecta ☺
- Toti algoritmii de join proceseaza doar doua tabele simultan (apoi rezultatul cu a treia, etc).
- Rezultatele de la un join sunt pasate in urmatoarea operatie join fara a fi stocate.
- Ordinea in care se efectueaza JOIN-ul influenteaza viteza de raspuns.[10, 30, 5, 60]
- QO incearca toate permutarile de JOIN.
- Cu cat sunt mai multe tabele, cu atat mai multe planuri de evaluat. [cate ?]

Join

- Cu cat sunt mai multe tabele, cu atat mai multe planuri de evaluat = O(n!)
- Nu este o problema cand sunt utilizati parametri dinamici [De ce ?]

Join – Nested Loops (anti patern)

- Ca si cum ar fi doua interogari: cea exterioara pentru a obtine o serie de rezultate dintr-o tabela si cea interioara ce preia fiecare rand obtinut si apoi informatia corespondenta din cea de-a doua tabela.
- Se pot folosi *Nested Selects* pentru a simula algoritmul de nested loops [latenta retelei, usurinta implementarii, Object-relational mapping (N+1 selects)].

Join – nested selects [PHP] java, perl on “luke…”

```
$qb = $em->createQueryBuilder();
$qb->select('e')
    ->from('Employees', 'e')
    ->where("upper(e.last_name) like :last_name")
    ->setParameter('last_name', 'WIN%');
$r = $qb->getQuery()->getResult();

foreach ($r as $row) {
    // process Employee
    foreach ($row->getSales() as $sale) {
        // process Sale for Employee
    }
}
```

Join – nested selects

Doctrine

Only on source code level—don't forget to disable this for production. Consider building your own configurable logger.

```
$logger = new \Doctrine\DBAL\Logging\EchoSqlLogger;
$config->setSQLLogger($logger);
```

Join – nested selects

Doctrine 2.0.5 generates N+1 `select` queries:

```
SELECT e0_.employee_id AS employee_id0 -- MORE COLUMNS
  FROM employees e0_
 WHERE UPPER(e0_.last_name) LIKE ?

    SELECT t0.sale_id AS SALE_ID1 -- MORE COLUMNS
      FROM sales t0
     WHERE t0.subsidiary_id = ?
       AND t0.employee_id = ?

    SELECT t0.sale_id AS SALE_ID1 -- MORE COLUMNS
      FROM sales t0
     WHERE t0.subsidiary_id = ?
       AND t0.employee_id = ?
```

Join – nested selects

- DB executa joinul exact ca si in exemplul anterior. Indexarea pentru nested loops este similara cu cea din selecturile anterioare:

1. Un FBI (function based Index) UPPER(last_name)
2. Un Index concatenat peste subsidiary_id, employee_id

Join – nested selects

- Totusi, in BD nu avem latenta din retea.
- Totusi, in BD nu sunt transferate datele intermediare (care sunt *piped* in BD).

- Pont: executati JOIN-urile in baza de date si nu in Java/PHP/Perl sau in alt limbaj (ORM).

Join – nested selects

- Cele mai multe ORM permit SQL joins.
- *eager fetching* – probabil cel mai important (va prelua si tabela vanzari –in mod join– atunci cand se interogheaza angajatii).
- Totusi *eager fetching* nu este bun atunci cand este nevoie doar de tabela cu angajati (aduce si date irelevante) – nu am nevoie de vanzari pentru a face o carte de telefoane cu angajatii.
- O configurare statica nu este o solutie buna.

```
$qb = $em->createQueryBuilder();
$qb->select('e,s')
   ->from('Employees', 'e')
   ->leftJoin('e.sales', 's')
   ->where("upper(e.last_name) like :last_name")
   ->setParameter('last_name', 'WIN%');
$r = $qb->getQuery()->getResult();
```

Doctrine 2.0.5 generates the following SQL statement:

```
SELECT e0_.employee_id AS employee_id0
       -- MORE COLUMNS
  FROM employees e0_
  LEFT JOIN sales s1_
         ON e0_.subsidiary_id = s1_.subsidiary_id
        AND e0_.employee_id = s1_.employee_id
 WHERE UPPER(e0_.last_name) LIKE ?
```

```
-----------------------------------------------------------------------
|Id |Operation                          | Name          | Rows | Cost |
-----------------------------------------------------------------------
| 0 |SELECT STATEMENT                   |               |  822 |   38 |
| 1 | NESTED LOOPS OUTER                |               |  822 |   38 |
| 2 |  TABLE ACCESS BY INDEX ROWID      | EMPLOYEES     |    1 |    4 |
|*3 |   INDEX RANGE SCAN                | EMP_UP_NAME   |    1 |      |
| 4 |  TABLE ACCESS BY INDEX ROWID      | SALES         |  821 |   34 |
|*5 |   INDEX RANGE SCAN                | SALES_EMP     |   31 |      |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  3 - access(UPPER("LAST_NAME") LIKE 'WIN%')
      filter(UPPER("LAST_NAME") LIKE 'WIN%')
  5 - access("E0_"."SUBSIDIARY_ID"="S1_"."SUBSIDIARY_ID"(+)
        AND  "E0_"."EMPLOYEE_ID"  ="S1_"."EMPLOYEE_ID"(+))
```

Join – nested selects

- Sunt bune daca sunt intoarse un numar mic de inregistrari.

- http://blog.fatalmind.com/2009/12/22/latency-security-vs-performance/

Join – Hash join

- Evita traversarea multipla a B-tree din cadrul inner-querry (din nested loops) construind cate un o tabela hash pentru inregistrarile candidat.
- *Hash join* imbunatatit daca sunt selectate mai putine coloane.
- A se indexa predicatele independente din where pentru a imbunatati performanta. (pe ele este construit hashul)

Join – Hash join

```
SELECT * FROM
sales s JOIN employees e
ON (s.subsidiary_id = e.subsidiary_id
  AND s.employee_id = e.employee_id )
WHERE s.sale_date > trunc(sysdate) -
  INTERVAL '6' MONTH
```

Join – Hash join

```
-------------------------------------------------------------------------
| Id | Operation           | Name      | Rows  | Bytes | Cost |
-------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |           | 49244 |    59M| 12049|
|* 1 |  HASH JOIN          |           | 49244 |    59M| 12049|
|  2 |   TABLE ACCESS FULL| EMPLOYEES | 10000 |     9M|   478|
|* 3 |   TABLE ACCESS FULL| SALES     | 49244 |    10M| 10521|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."SUBSIDIARY_ID"="E"."SUBSIDIARY_ID"
          AND "S"."EMPLOYEE_ID"  ="E"."EMPLOYEE_ID")
   3 - filter("S"."SALE_DATE">TRUNC(SYSDATE@!)
                             -INTERVAL'+00-06' YEAR(2) TO MONTH)
```

Join – Hash join

- **Indexarea predicatelor utilizate in join nu imbunatatesc performanta hash join !!!**
- Un index ce ar putea fi utilizat este peste sale_date
- Cum ar arata daca s-ar utiliza indexul ?

Join – Hash join

```
-------------------------------------------------------------------
| Id | Operation                    | Name       | Bytes| Cost|
-------------------------------------------------------------------
|  0 | SELECT STATEMENT             |            |   59M| 3252|
|* 1 |  HASH JOIN                   |            |   59M| 3252|
|  2 |   TABLE ACCESS FULL          | EMPLOYEES  |    9M|  478|
|  3 |   TABLE ACCESS BY INDEX ROWID| SALES      |   10M| 1724|
|* 4 |    INDEX RANGE SCAN          | SALES_DATE |      |     |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."SUBSIDIARY_ID"="E"."SUBSIDIARY_ID"
          AND "S"."EMPLOYEE_ID"  ="E"."EMPLOYEE_ID"  )
   4 - access("S"."SALE_DATE" > TRUNC(SYSDATE@!)
                              -INTERVAL'+00-06' YEAR(2) TO MONTH)
```

Join – Hash join

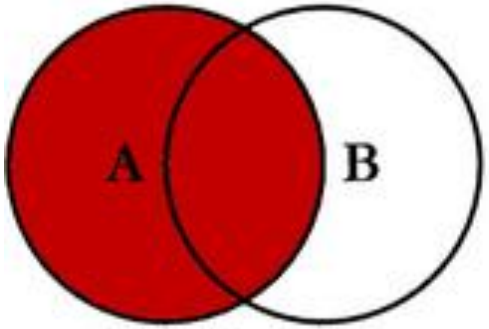
- Ordinea conditiilor din join nu influenteaza viteza (la nested loops influenta).

Bibliografie (online)

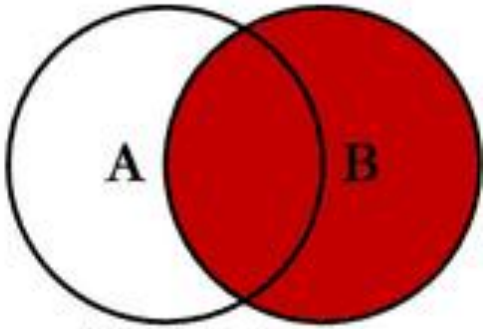
- http://use-the-index-luke.com/

  (puteti cumpara si cartea in format PDF – dar nu contine altceva decat ceea ce este pe site)

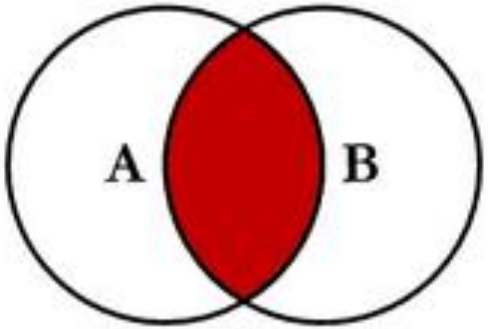
SQL JOINS



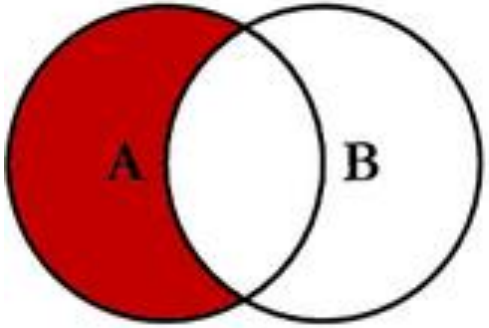

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



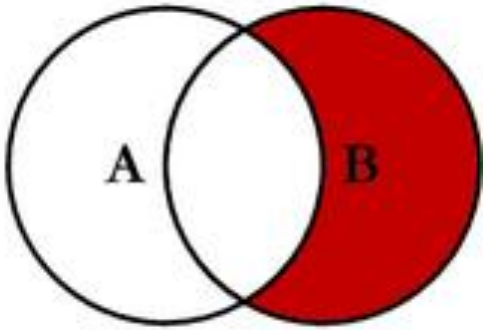

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



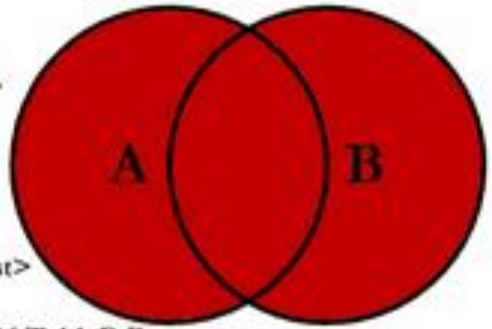

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



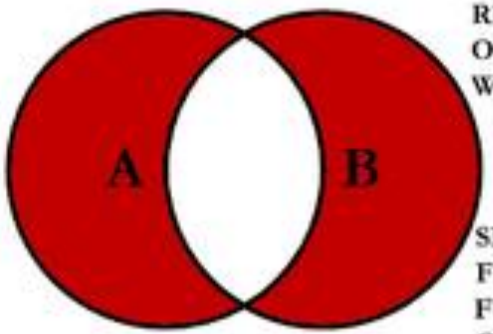

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```