

Practică SGBD

A big party is being held to honor relational database systems and their impact on modern society. Outside the venue, the host awaits the guests.

The first limousine arrives and out steps Oracle followed by 4 people.

Host: Who have you brought along?

Oracle: I have 4 DBA's in tow. One to install me, one to design the databases, one to administer me, and the other to justify the cost.

A second limo arrives and out steps DB2 followed by 40 people.

Host: Who have you brought along?

DB2: I have 2 DBA's, 2 hardware specialists, and 36 consultants.

A third limo arrives and out steps SQL Server all on his own.

Host: Why haven't you brought anyone?

SQL Server: I didn't bring anyone because I am easy to install and am basically self managing. But I did bring the #sqlhelp Twitter hashtag for when the [bip] hits the fan.

20 minutes later, up rushes MySQL, unshaved, hair a mess.

Host: Where have you been MySQL?

MySQL: Sorry, I thought it was February 31st.

Ce învățăm la acest curs ?

- Fiind vorba de practică... nimic (prea) teoretic.
- La curs sunt acoperite topicile:
 - cum optimizăm o interogare (indexare);
 - tranzacții (o bucată ramasă de la BD);
 - NoSQL – o scurtă introducere.
- La laborator: facem teme în PL/SQL și un proiect în ce vrem noi dar care să foloseasca serverul Oracle (dacă tot facem PL/SQL...).

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



MARKUS WINAND

Syntactic & Semantic

- Putem considera o interogare SQL ca fiind o propoziție din engleză ce ne indică ce trebuie făcut fără a ne spune cum este făcut:

```
SELECT prenume  
      FROM studenti  
      WHERE nume = ' POPESCU '
```

... și dacă am un timp de răspuns de 15 secunde ?

La baza unei aplicații ce nu merge stau două greșeli umane*

- Autorului unei interogări SQL nu îi pasă (de obicei) ce se întâmplă “în spate”.
- Autorul interogării nu se consideră vinovat dacă timpul de răspuns al SGBD-ului este mare (evident, cel care l-a inventat nu prea a știut ce face).
- Soluția ? Simplu: nu mai folosim Oracle, trecem pe MySQL, PostgreSQL sau SQL Server (că ne-a zis nouă cineva că merge mai bine).

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

De fapt...

- Singurul lucru pe care dezvoltatorii trebuie să îl învețe este cum să indexeze corect (bine, poate nu singurul...).
- Cea mai importantă informație este felul în care aplicația va utiliza datele.
- Traseul datelor nu este cunoscut nici de client, nici de administratorul bazei de date și nici de consultanții externi; singurul care știe acest lucru este dezvoltatorul aplicației !

...cuprins... (legat de indexare)

- Anatomia unui index
- Clauza WHERE
- Performanță și Scalabilitate
- JOIN
- Clustering
- Sortare & grupare
- Rezultate parțiale
- INSERT, UPDATE, DELETE

Anatomia unui index

- *“An index makes the query fast”* - cât 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 ("nume"=' POPESCU ')

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("nume"='POPESCU')

Anatomia unui index

- Un index este o structură* distinctă într-o bază de date ce poate fi construită utilizând comanda **create index**.
- Are nevoie de propriul spațiu pe HDD și *poartează* tot către informațiile aflate în baza de date (la fel ca și cuprinsul unei cărți, redundantă până 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

- Căutarea după un index este asemănătoare cu căutarea într-o carte de telefon.
- Indexul din BD trebuie să fie mult mai optimizat din cauza dinamicității unei BD
[insert / update / delete]
- Indexul trebuie menținut fără a muta cantități mari de informație.

Timpul necesar căutării într-un fișier “sortat”

- Sa presupunem că avem 1.000.000 date de “dimensiune egală” [din acest motiv old SGBD gen FoxPro nu prea se împacă bine cu variabile ca cele de tip varchar2].
- Căutarea binară $\Rightarrow \log_2(1.000.000) = 20$ citiri
- Un HDD de 7200RPM face o rotație completă în $60/7200 = 0.008333..$ = 8.33ms
- Pentru un Seagate ST3500320NS, track-to-track seek time = 0.8ms

Timpul necesar căutării într-un fișier “sortat”

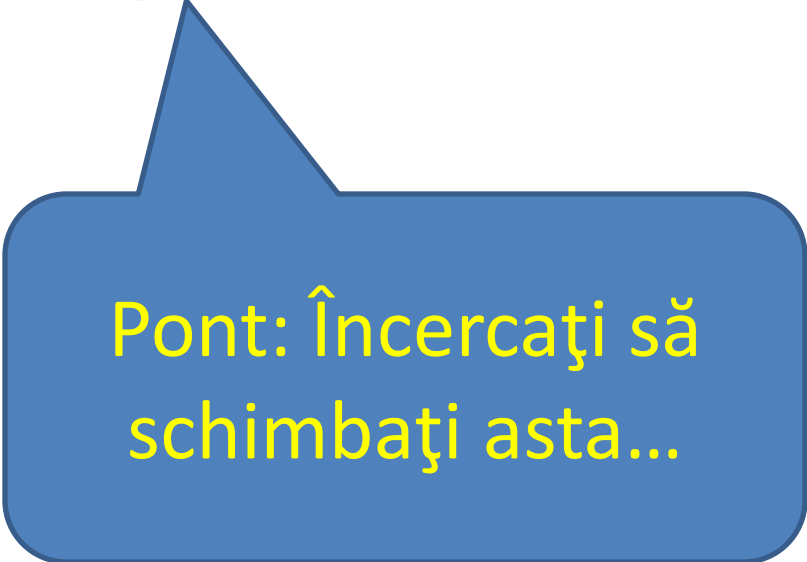
- Căutarea (de 0.8ms) și citirea unei piste (8.33ms) să zicem că ajungem pentru o citire la 10ms.
- 20 citiri = 200ms = 0.2”
- Probabil timpul este mare pentru că ultimele informații se pot afla pe aceeași pistă ceea ce va “eficientizează” și nu vom mai citi aceeași pistă ultimele 3-4 ture => 0.16”.

Timpul necesar căutării într-un fișier “sortat”

- Dacă ar fi trebuit să caut 10 valori ? Ar fi fost necesare 2 secunde
- Dacă ar fi fost necesar să caut 100 de valori (care de obicei sunt afișate pe o pagina web gen e-bay).... 20 de secunde. Sigur nu v-ar plăcea să așteptați un magazin online 20 de secunde până să vă afișeze cele 100 de produse ;) ... și OLX merge mai bine :D
- Nici nu vreau să mă gândesc ce s-ar întâmpla dacă datele nu ar fi sortate...

Cum obțin timpi și mai mici ?

$$\log_2(1.000.000)$$



Pont: Încercați să
schimbați asta...

Anatomia unui index

- Cum funcționează ?
 - pe baza unei liste dublu înlănțuite
 - pe baza unui arbore de cautare
- Prin intermediul listei se pot insera cantități mari de date fără a fi nevoie să le deranjăm pe cele existente.
- Arborele este utilizat pentru a căuta datele indexate (B⁺-trees)

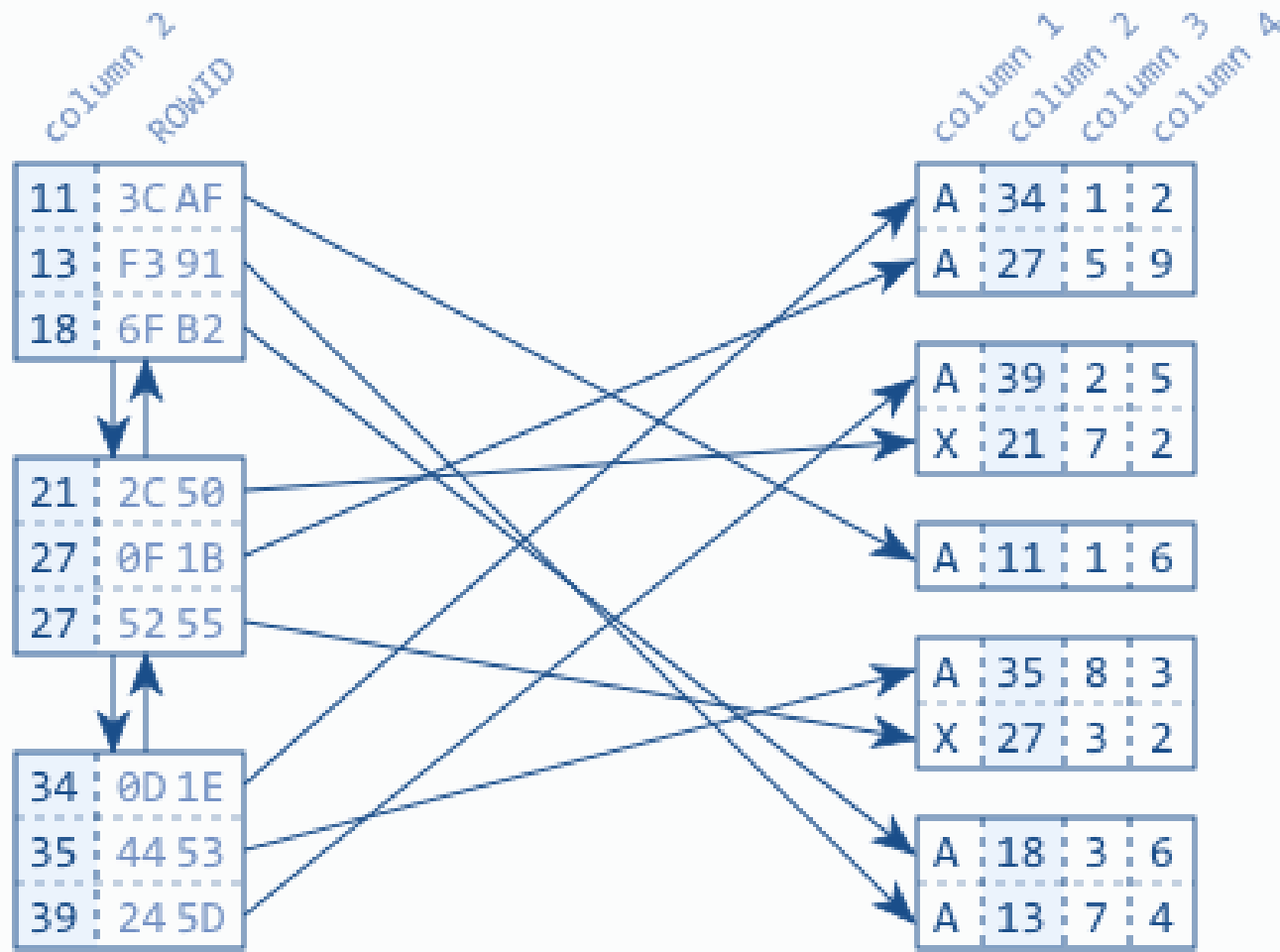
Anatomia unui index

- Bazele de date utilizează listele pentru a înlănțui “frunzele” arborelui menționat
- Fiecare bloc al unui index are o aceeași dimensiune, conține câte un pointer către următoarea frunză și unul către o linie a unui tabel.

Anatomia unui index

Index Leaf Nodes
(sorted)

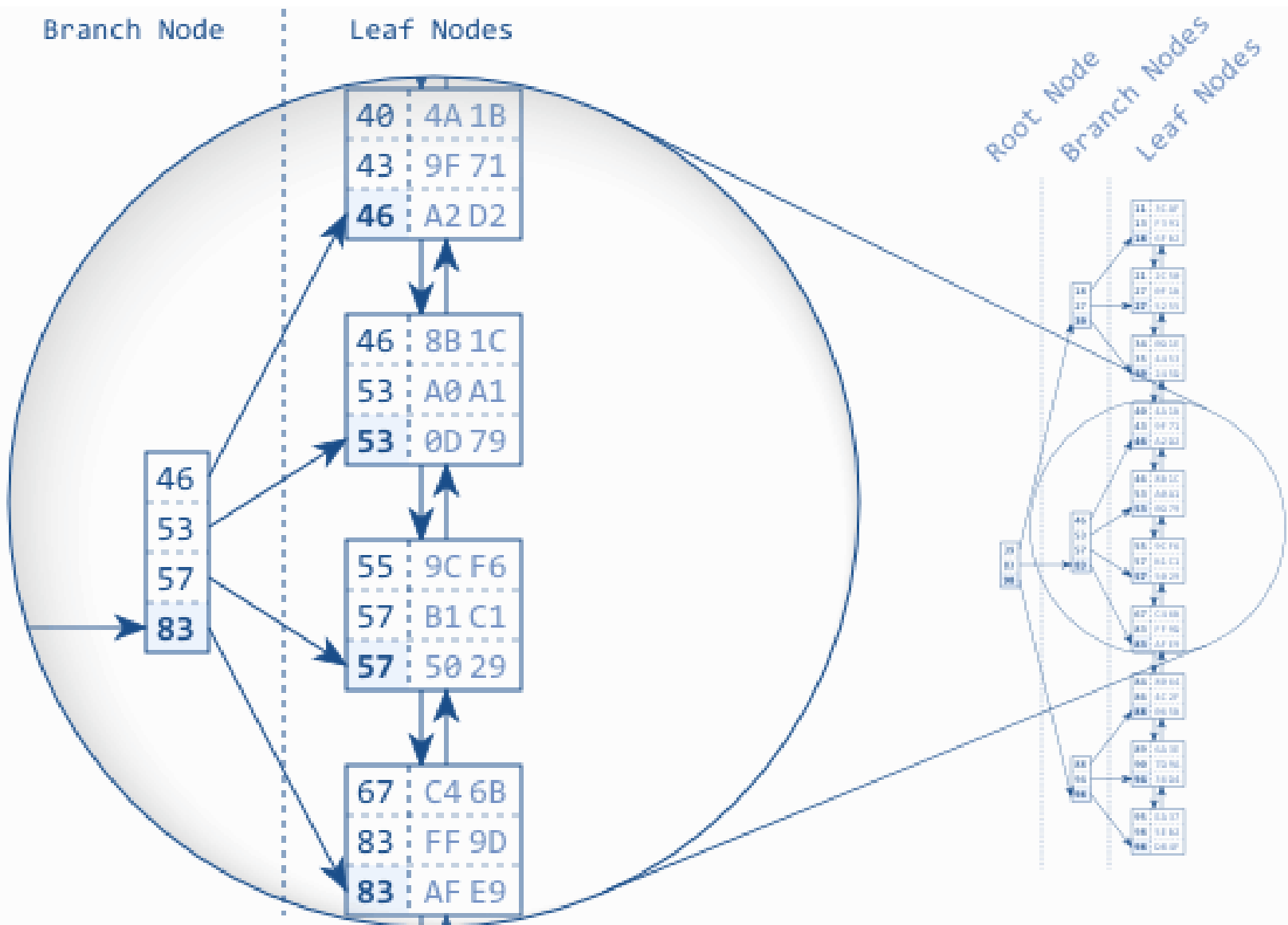
Table
(not sorted)



Anatomia unui index

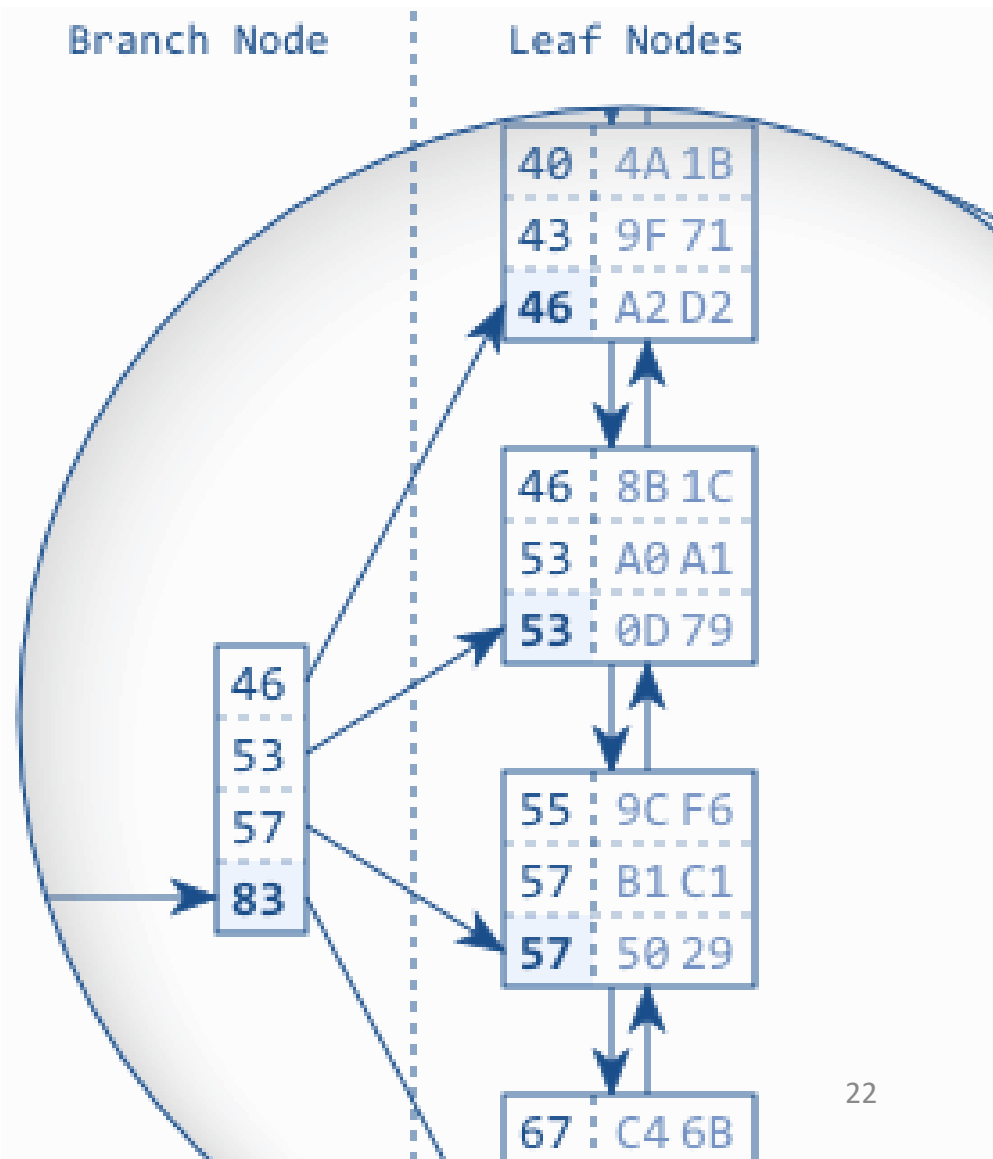
- “frunzele” nu sunt stocate pe disc în ordine sau având o aceeași distribuție – poziția pe disk nu corespunde cu ordinea logică a indecșilor (de exemplu, dacă indexăm mai multe numere între 1 și 100 nu e neapărat ca 50 să se afle exact la mijloc) – putem avea 80% de valori 1.
- SGBD-ul are nevoie de cea de-a doua structură pentru a căuta rapid între indecșii amestecați.

Anatomia unui index

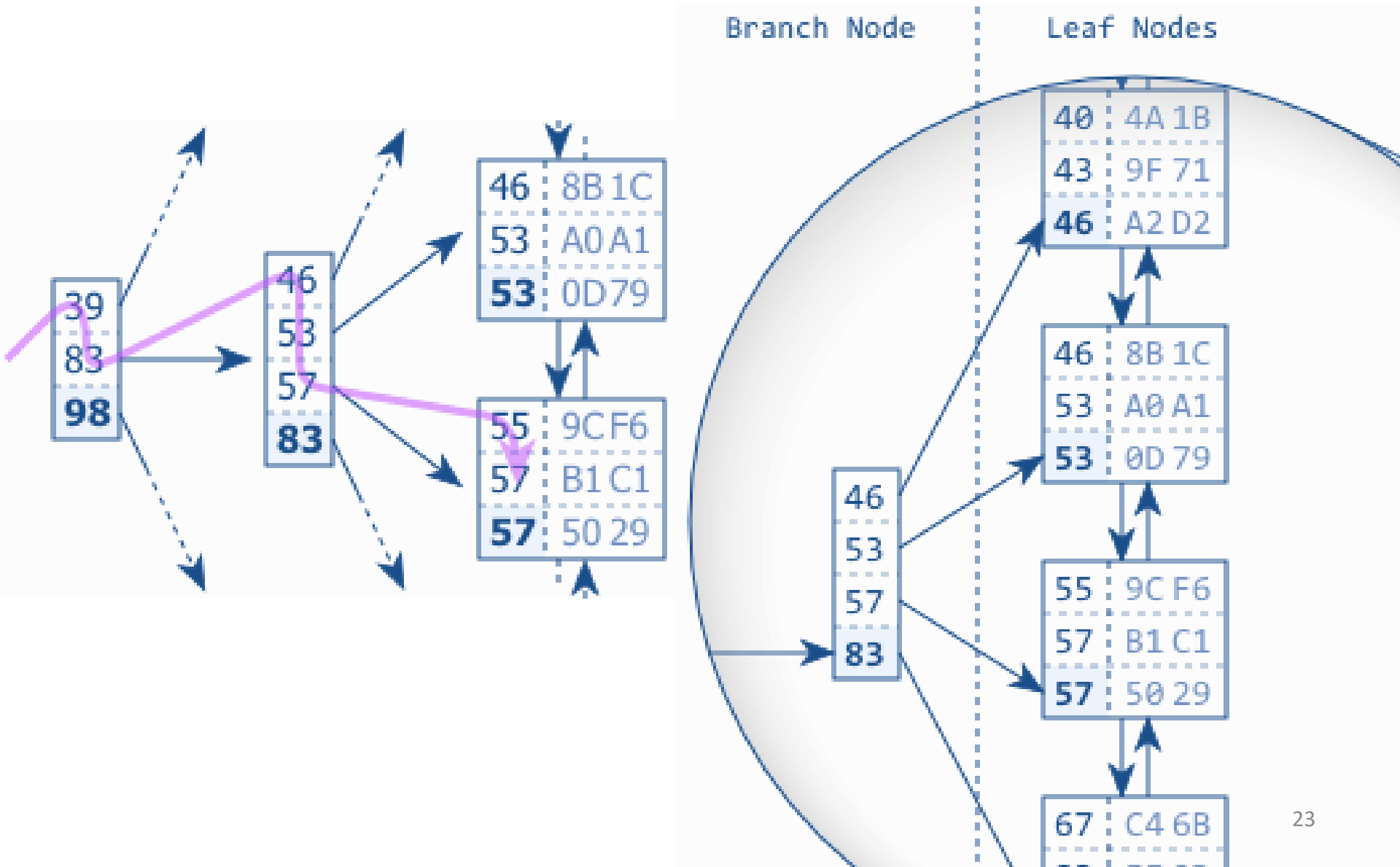


Anatomia unui index

Pointerul către următorul nivel indică cea mai mare valoare a acestui următor nivel.



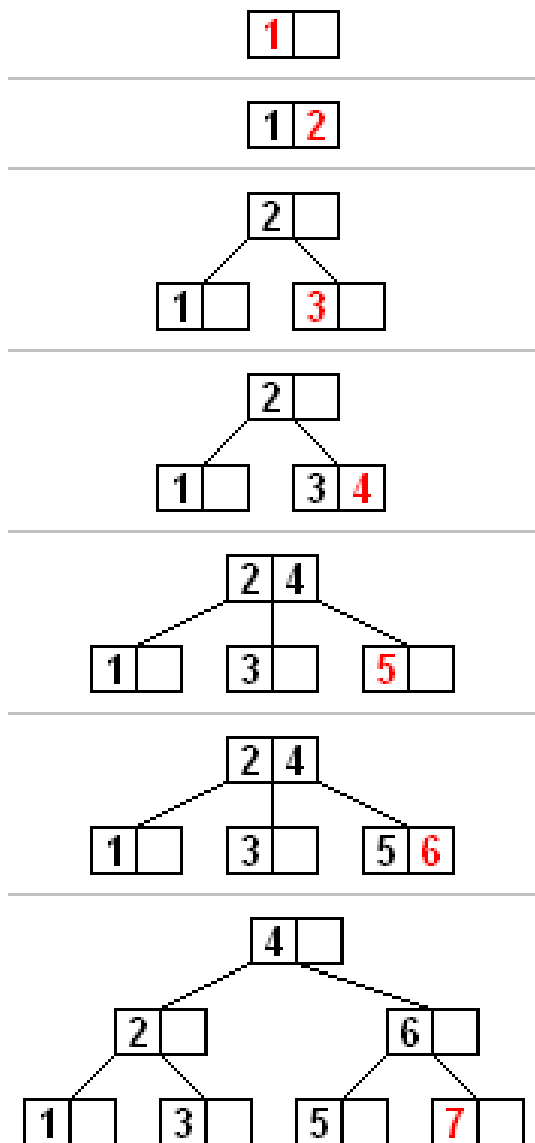
Anatomia unui index



Anatomia unui index

- Un *B-tree* este un arbore echilibrat !
- Un *B-tree* nu este un arbore binar !
- Adâncimea arborelui este identică spre oricare dintre frunze.
- Odată creat, baza de date menține indecșii în mod automat, indiferent de operația efectuată asupra bazei de date (insert/delete/update)
- *B-tree*-ul facilitează accesul la o frunză;
- Cât de repede ? *[first power of indexing]*

Cum se balansează un B-tree ?



- Dacă ar avea maxim 2 locații pe nod ar putea să ajungă să se comporte ca un arbore binar...
- Având mai multe locații și fiind “sparse”, e mult mai flexibil (observați că uneori rămâne echilibrat chiar după inserare)
- Dacă ar avea mai multe locații libere în fiecare nod, nevoia de echilibrare ar fi și mai rară.

Cum se balansează un B-tree ?

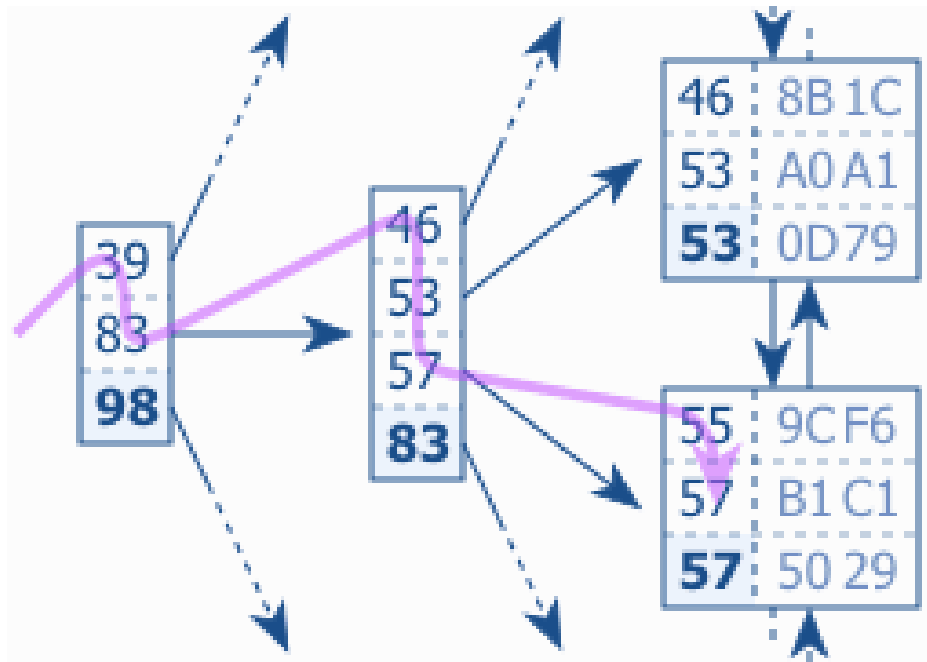
- Nu ne interesează la nivel formal (pentru asta aveți cursuri de algoritmică/programare etc.)
- Ideea de bază este ca atunci când se ajunge la numărul maxim de valori într-un nod, el se scindează în două noduri și se reface echilibrarea, când este eliminată ultima valoare, se reechilibrează în sens invers.
- Echilibrarea nu e neapărat să ajungă până în rădăcină ea putându-se face în valorile libere de până la rădăcină.

Anatomia unui index

- Deși găsirea informației se face în timp logaritmic, există “mitul” că un index poate degenera (și ca soluție este “reconstruirea indexului”). - fals deoarece arborele se autobalansează.

Anatomia unui index

- De ce ar funcționa un index greu ?



- Atunci când sunt mai multe rânduri (57,57...)

Anatomia unui index

- De ce ar funcționa un index greu ?
- După găsirea indexului corespunzător, trebuie obținut rândul din tabelă
- Căutarea unei înregistrări indexate se face în 3 pași:
 - Traversarea arborelui [limita superioară: adâncimea arborelui: oarecum rapid]
 - Căutarea frunzei în lista dublu înlănțuită [încet]
 - Obținerea informației din tabel [încet]

Anatomia unui index

- Este o concepție greșită să credem că arborele s-a dezechilibrat și de asta căutarea este înceată. În fapt, traversarea arborelui pare să fie cea mai rapidă.
- Dezvoltatorul poate “întreba” baza de date despre felul în care îi este procesată interogarea.

Anatomia unui index

- În Oracle există trei tipuri de operații importante:

INDEX UNIQUE SCAN

INDEX RANGE SCAN

TABLE ACCESS BY INDEX ROWID

- Cea mai costisitoare este INDEX RANGE SCAN.
- Dacă sunt mai multe rânduri, pentru fiecare dintre ele va face TABLE ACCESS – în cazul în care tabela este imprăștiată în diverse zone ale HDD, și această operație devine greoaie.

Planul de executie

- Pentru a interoga felul în care Oracle procesează o interogare: **EXPLAIN PLAN FOR**

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

- Pentru a afișa rezultatul, se execută
select * from table(dbms_xplan.display) ;

Planul de executie

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

```
PLAN_TABLE_OUTPUT
```

```
-----  
Plan hash value: 3461732445
```

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	61	3 (0)	00:00:01
* 1	TABLE ACCESS FULL	USERS	1	61	3 (0)	00:00:01

```
-----
```

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

```
-----  
PLAN_TABLE_OUTPUT
```

```
-----  
1 - filter("NAME" LIKE 'Diana%')
```

```
13 rows selected.
```

Planul de executie

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

```
PLAN_TABLE_OUTPUT
```

```
-----  
-----  
Plan hash value: 203565131
```

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	61	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	USERS	1	61	2 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	USER_NAMES	1		1 (0)	00:00:01

```
-----
```

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

```
-----  
2 - access("USERNAME"='diana.lucaci')
```

```
14 rows selected.
```

Sau, versiunea SQL Developer...

The screenshot displays the Oracle SQL Developer interface with the 'student' database selected. The 'Connections' pane on the left shows a tree of tables under the 'student' connection, including 'USERS'. The 'Worksheet' tab is active, showing a SQL query: `SELECT * FROM USERS WHERE USERNAME LIKE 'diana.lucaci';`. The 'Query Result' tab is also open, displaying the execution plan for the query. The execution plan shows a 'SELECT STATEMENT' with a 'TABLE ACCESS (BY INDEX ROWID)' on the 'USERS' table, which is accessed via a 'UNIQUE SCAN' on the 'USER_NAMES' index. The 'Access Predicates' section shows the filter `USERNAME='diana.lucaci'`. The 'Messages - Log' pane at the bottom shows the status 'Messages', 'Logging Page', and 'Statements'.

Oracle SQL Developer : student

File Edit View Navigate Run Source Team Tools Window Help

Connections

- bd2_app
- local
- ServerTestPSGBD
- startrek
- student
 - Tables (Filtered)
 - ANSWERS
 - CHAPTERS
 - DELETED_QUESTIONS
 - QUESTION_CACHE
 - QUESTIONS
 - REPORTS
 - USERS
 - ID
 - NAME
 - USERNAME

Start Page student USERS

Worksheet Query Builder

```
1 SELECT * FROM USERS WHERE USERNAME LIKE 'diana.lucaci';
```

Query Result Explain Plan

SQL | 0,058 seconds

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	2
TABLE ACCESS (BY INDEX ROWID)	USERS	1	2
INDEX (UNIQUE SCAN)	USER_NAMES	1	1
Access Predicates			
USERNAME='diana.lucaci'			
Other XML			

Messages - Log

Messages Logging Page Statements

Line 1 Column 56 | Insert | Modified | Windows: CR

Clauza WHERE

Clauza **WHERE**

- Clauza **WHERE** dintr-un select definește condițiile de căutare dintr-o interogare SQL și poate fi considerată nucleul interogării – din acest motiv **influențează cel mai puternic** rapiditatea cu care sunt obținute datele.
- Chiar dacă **WHERE** este cel mai mare dușman al vitezei, de multe ori este “aruncat” doar “pentru că putem”.
- Un **WHERE** scris rău este principalul motiv al vitezei mici de răspuns 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 creat pe un primary key poate avea range scan dacă este interogată cu egalitate ?

Concatenarea indecșilor

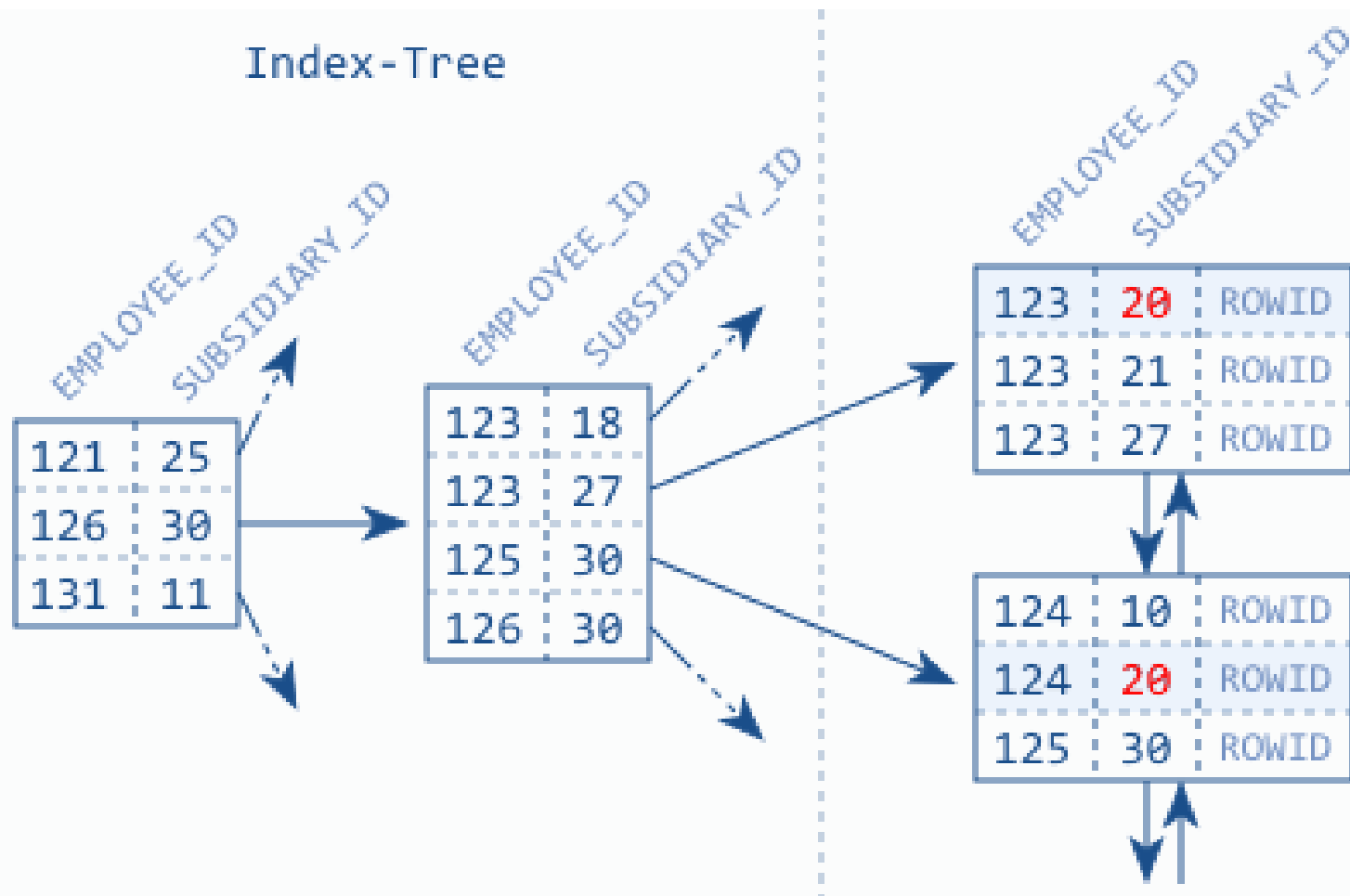
- Uneori este nevoie ca indexul să îl construim peste mai multe coloane

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

- După ce compania a fost cumpărată de o altă firmă (cu 9000 de angajați), încercăm iarași să găsim angajatul.

[sau ne putem gândi cum căutăm în cartea de telefon ... dacă mai folosește cineva așa ceva]

Concatenarea indecșilor



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 când o interogare folosește întreaga cheie (ambele câmpuri constituyente), căutarea este de tipul INDEX UNIQUE SCAN (dacă cele două formează împreună o cheie unică)
- Ce se întâmplă dacă vrem să căutăm doar după unul din câmpuri ? De exemplu după **subsidiary_id** ?

Clauza **WHERE**

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

- Ce se va întâmpla ?

Clauza **WHERE**

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

- Ce se va întâmpla ?
- Același lucru care s-ar întâmpla dacă v-aș pune sa îi căutați pe toți abonații telefonici cu prenumele Vasile.

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.

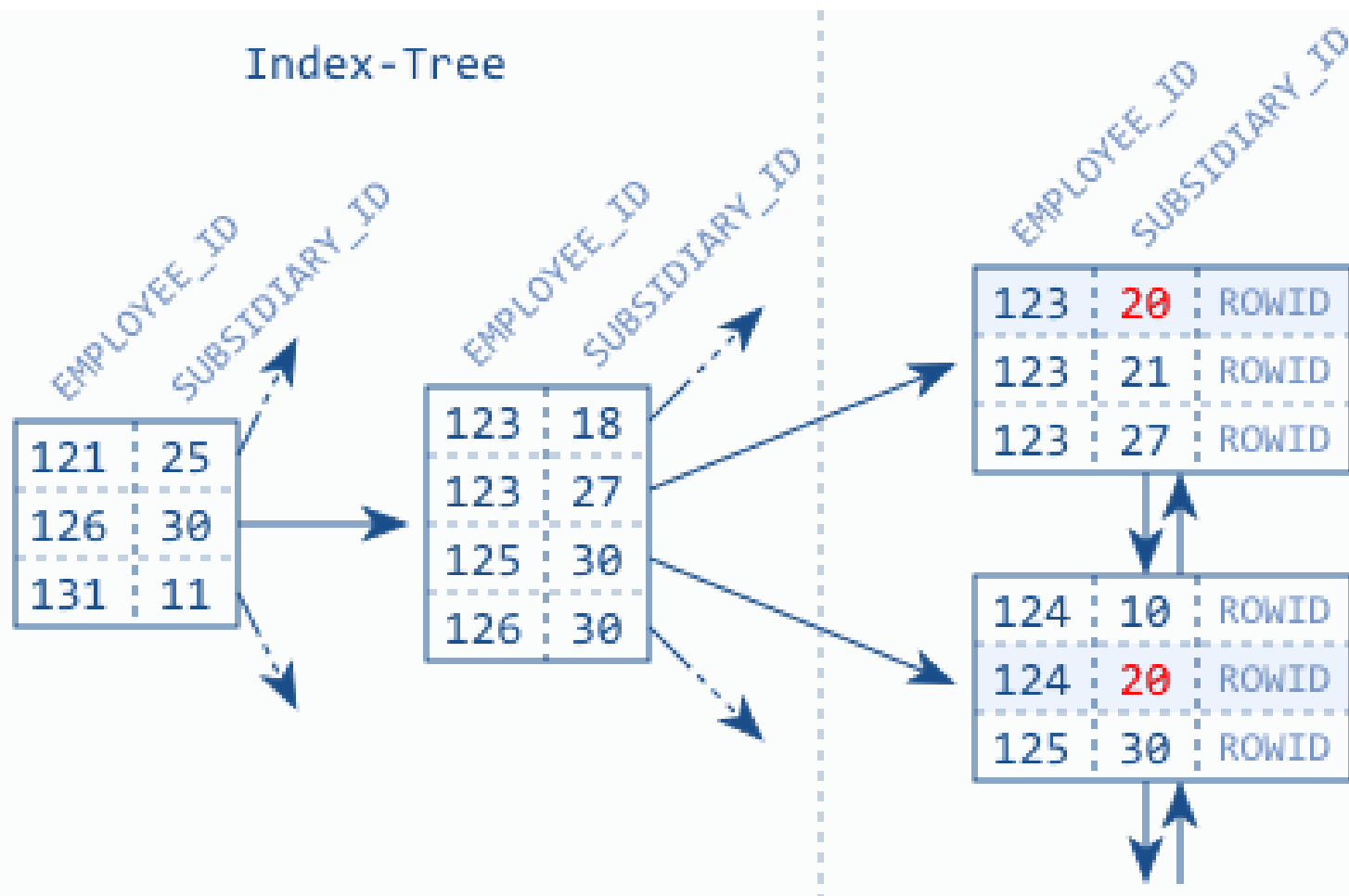
Cu cât a crescut timpul de execuție ?

Operația este rapidă într-un exemplu mic dar foarte costisitoare în caz contrar.

Clauza **WHERE**

- Uneori scanarea completă a bazei de date este mai eficientă decât accesul prin indecși. Acest lucru este parțial chiar din cauza timpului necesar căutării indecșilor.
- O singură coloană dintr-un index concatenat nu poate fi folosită ca index (excepție face prima coloana).

Concatenarea indecșilor – 1st column ?



Căutare după employee_id

Utilizarea doar a primei coloane din index este încă posibilă:

```
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 observă că valoarea 20 pentru **subsidiary_id** este distribuită aleator prin toata tabela. Din acest motiv, nu este eficient să căutăm după acest index.
- Cum facem ca să căutăm eficient?

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

- În continuare indexul este format din aceleași două coloane (dar în altă ordine).

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 când definim indecși concatenați este să stabilim ordinea.
- Dacă vrem să utilizăm 3 câmpuri pentru concatenare, căutarea este eficientă pentru câmpul 1, pentru 1+2 și pentru 1+2+3 dar nu și pentru alte combinații.

Clauza **WHERE**

- Atunci când este posibil, este de preferat utilizarea unui singur index (din motive de spațiu ocupat pe disc și din motive de eficiență a operațiilor ce se efectuează asupra bazei de date).
- Pentru a face un index compus eficient trebuie ținut cont și care din câmpuri ar putea fi interogate independent – acest lucru este știut de obicei **doar de către programator**.

Indecși “înceți”

- Schimbarea indecșilor poate afecta întreaga bază de date ! (operațiile pe aceasta pot deveni mai greoaie din cauză că managementul lor este făcut diferit)
- Indexul construit anterior este folosit pentru toate interogările în 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)

- Dacă avem doi indecși disjuncți și în clauza WHERE sunt folosiți ambii ? Pe care dintre ei îi va considera BD? Este mereu eficient să se țină cont de indecși ?

Indecși “înceți”

- ***The Query Optimizer***
- Componenta ce transformă interogarea într-un plan de execuție (aka compiling / parsing).
- Două tipuri de optimizare:
 - *Cost based optimizers* (CBO) – mai multe planuri, calculează costul lor și rulează pe cel mai bun;
 - *Rule-based optimizers* (RBO) – folosește un set de reguli hardcodat (de DBA).

CBO poate sta prea mult să caute prin indecși și RBO să fie mai eficient în acest caz [1000x1000 tbl]

Indecși “înceți”

- Cum ați executa această interogare dacă ați fi QO (indexul a rămas 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 observă că nu este utilizat indexul, QO preferând să parcurgă toată tabela și să filtreze rândurile corespunzătoare [și era doar un singur rând căutat].
- Un exemplu chiar mai surprinzător:

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)

Și fără a modifica în nici un fel tabela sau indecșii implicați (ci doar valoarea căutată: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 întâmplă acest lucru ?

Statistici

- CBO utilizează statistici despre BD (de ex. privind: tabelele, coloanele, indecșii). De exemplu, pentru o **tabelă** poate memora:
 - valoarea maximă/minimă,
 - numărul de valori distincte,
 - numărul de câmpuri NULL,
 - distribuția datelor (histograma),
 - dimensiunea tabelului (nr rânduri/blocuri).

Statistici

- CBO utilizeaza statistici despre BD (de ex. privind: tabelele, coloanele, indecșii). De exemplu, pentru un **index** poate memora:
 - adâncimea B-tree-ului,
 - numărul de frunze,
 - numărul de valori distincte din index,
 - factor de “*clustering*” (date situate pe aceeași pistă pe HDD sau în piste apropiate).
- **Utilizarea indecșilor nu e mereu soluția cea mai potrivită.**

Indecsi bazati pe functii

Applies to
DB2 
MySQL 
Oracle 
PostgreSQL 
SQL Server 

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 informatiile in indecsi.

Funcții

```
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 | 36000 |      18   (0)| 00:00:01 |
|*  1  |   TABLE ACCESS FULL            | EMPLOYEES      |    100 | 36000 |      18   (0)| 00:00:01 |
-----

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

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

13 rows selected.
```

Isi da seama ca e mai eficient sa evalueze
functia pentru valoarea constanta si sa nu faca
acest lucru pentru fiecare rand in parte.

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'
```

Funcții

- 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. [days untill xmas]

Funcții

- Nu există cuvinte rezervate sau optimizări pentru F# (altele decât cele deja explicate).
- Uneori instrumentele pentru *Object relation mapping* (ORM tools) injectează din prima o funcție de conversie a tipului literelor (upper / lower). De ex. Hibernate convertește totul în lower.
- Puteti construi proceduri stocate deterministe ca sa fie folosite in F#. **getAge ?!?!**

Functii – nu indexati TOT

- De ce ? (nu are sens sa fac un index pt. *lower*) (daca tot aveti peste upper). De fapt, daca exista o functie bijectiva de la felul in care sunt indexate datele la felul in care vreti sa interogati baza de date, mai bine refaceti interogarea – cu siguranta este posibil !).
- 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*

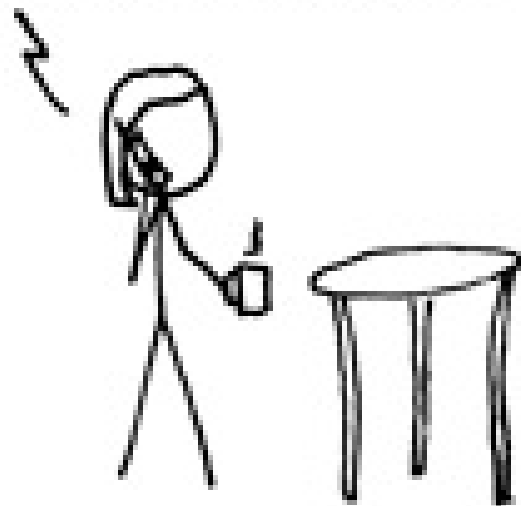
```
statement = "SELECT * FROM users  
            WHERE name ='" + userName + "';"
```

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

HI, THIS IS
YOUR SON'S SCHOOL.
WE'RE HAVING SOME
COMPUTER TROUBLE.

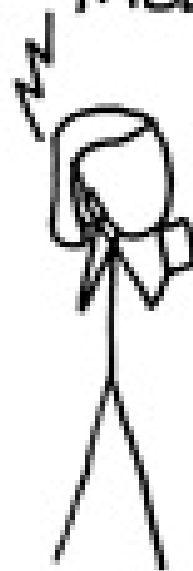


OH, DEAR - DID HE
BREAK SOMETHING?

IN A WAY -

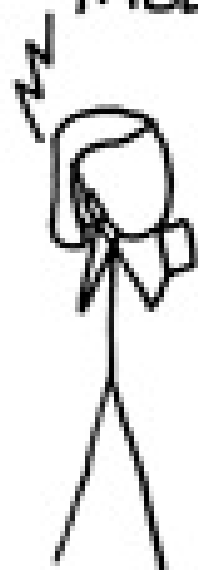


DID
NAME
Robert
TABL



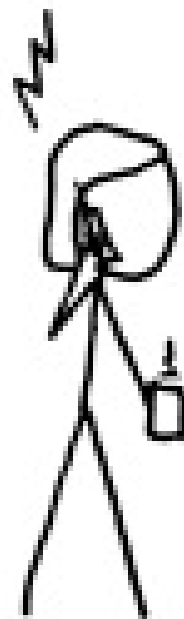
E
G?

DID YOU REALLY
NAME YOUR SON
Robert'); DROP
TABLE Students;-- ?



OH, YES. LITTLE
BOBBY TABLES,
WE CALL HIM.

WELL, WE'VE LOST THIS
YEAR'S STUDENT RECORDS.
I HOPE YOU'RE HAPPY.



AND I HOPE
YOU'VE LEARNED
TO SANITIZE YOUR
DATABASE INPUTS.

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 interogariile 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 compiler:
 - 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) – esemplu 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();
```

Se repeta pentru
fiecare parametru

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');
```

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')
```



Inceput



Sfarsit

Cautari pe intervale

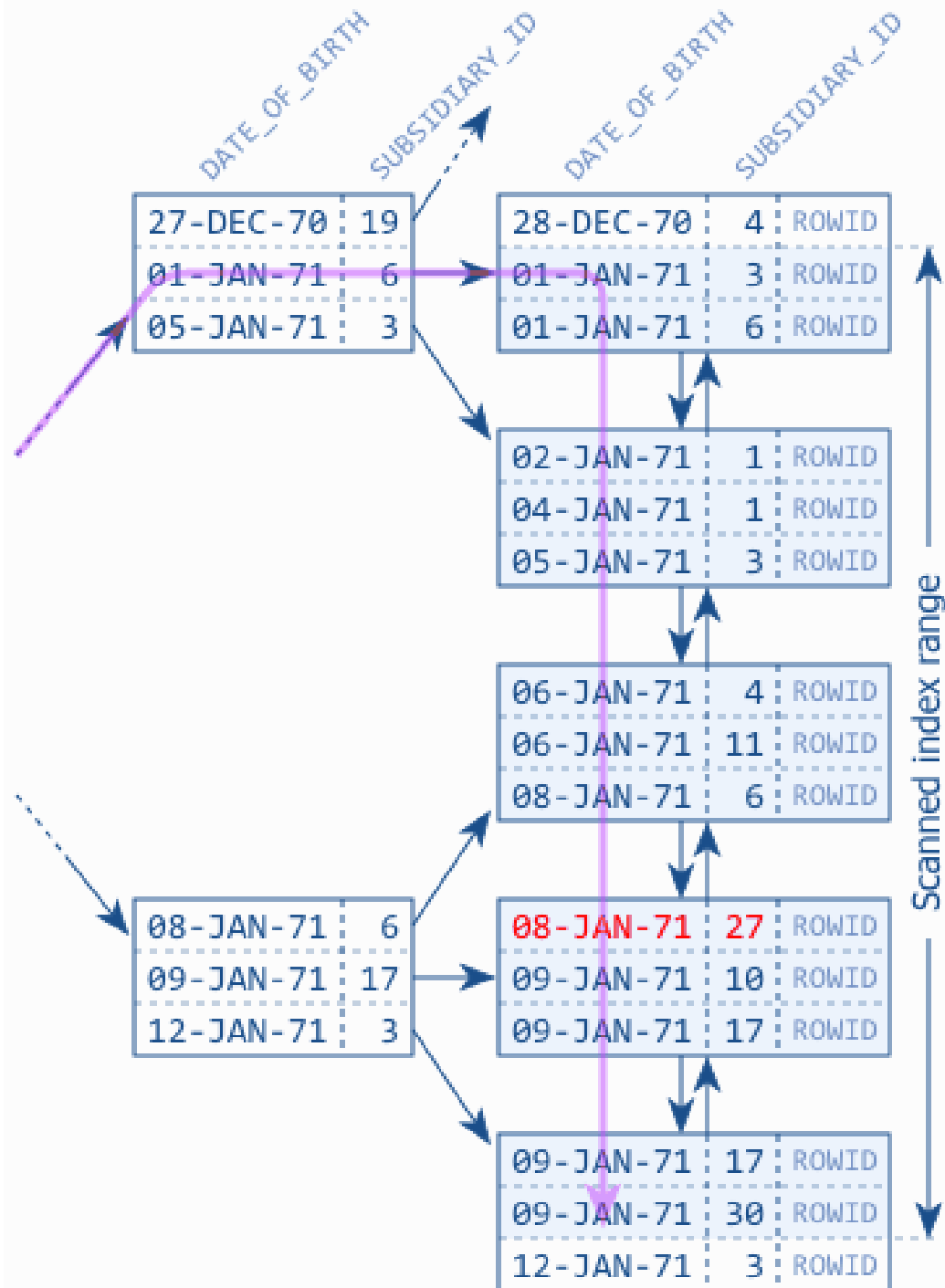
```
SELECT first_name, last_name,  
       date_of_birth  
FROM employees  
WHERE  
       date_of_birth >= TO_DATE(?, 'YYYY-  
MM-DD') AND  
       date_of_birth <= TO_DATE(?, 'YYYY-  
MM-DD') AND  
AND subsidiary_id = ?
```



???

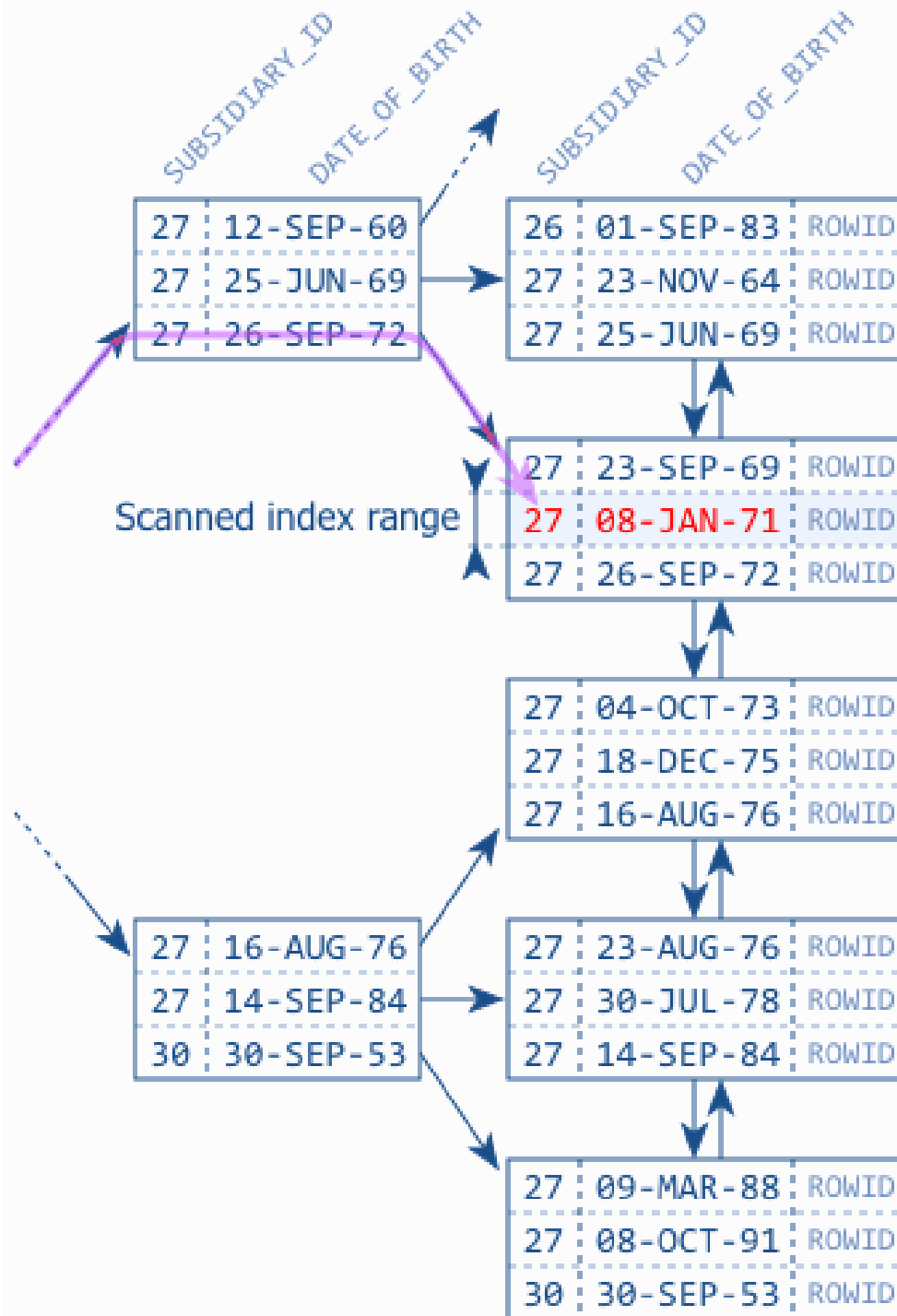
Cautari pe intervale

- 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 interogari in functie in care caracterul % este asezat in diverse pozitii:

LIKE

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

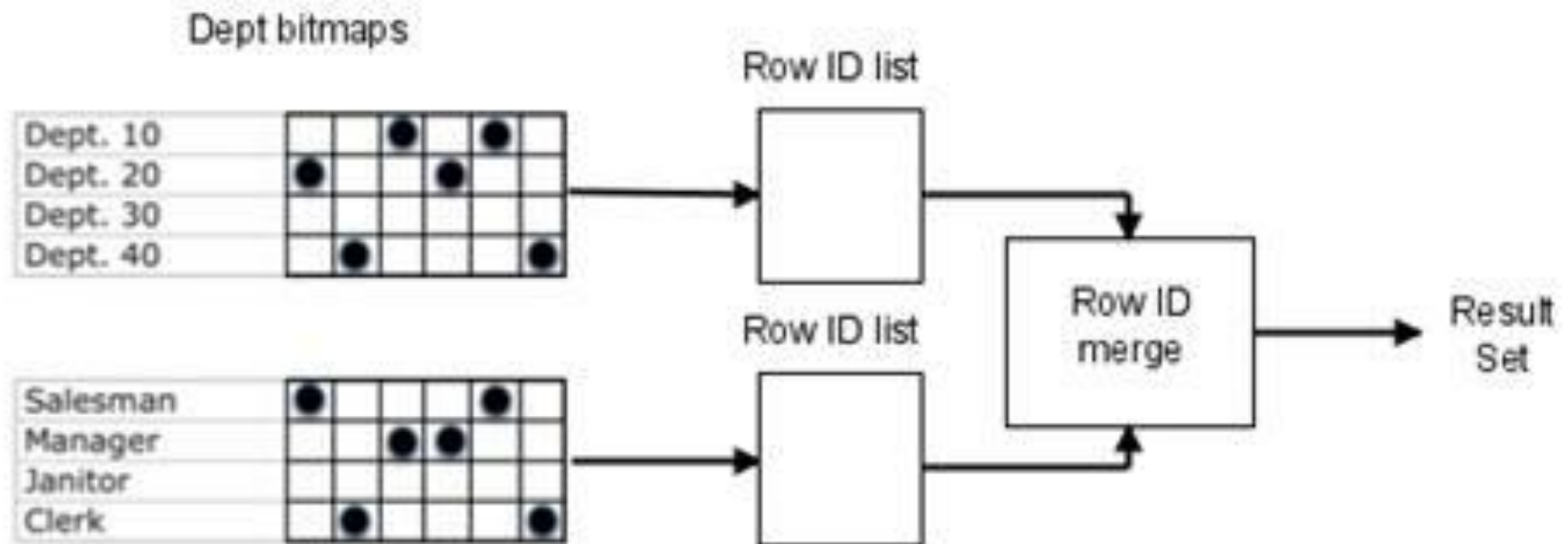
Contopirea indecsilor

- 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 interogările 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

Applies to
DB2
MySQL
Oracle
PostgreSQL
SQL Server

- Sa analizam interogarea:

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

- Preia toate mailurile nevizualizate (de exemplu). 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 ( ?, ?, ?, ?, ? );
```

Neinserand data de
nastere, aceasta va fi
NULL

- Noul rand nu va fi indexat:

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

Table
access
full

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

Indexarea NULL in Oracle

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



NOT NULL

- 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');
```

Asta este 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	

0		SELECT STATEMENT			
*		TABLE ACCESS FULL		EMPLOYEES	

- 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;
```

In opinia lui,
ambele pot fi
NULL.

Desi id este
NOT NULL

```
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

li 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;
```

Daca initial last_name este nenul va sti
ca upper(last_name) este tot nenul.

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

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

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 (indexul este peste sirul 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
```

Va merge pe index ? (care era construit peste sirul de caractere ?!?!)

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 ultima 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 interogările pentru perioada ca și condiții explicite (chiar dacă e vorba de o singură 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:

Index peste sale_date

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

Fie direct sir de caractere sau chiar parametru
dinamic trimis ca sir de caractere.

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 > ?
```

Chiar de are index peste numeric_number, nu are peste suma lui cu 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 interogările 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

**Don't ask a DBA to help you move furniture.
They've been known to drop tables...**

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 = ?
```

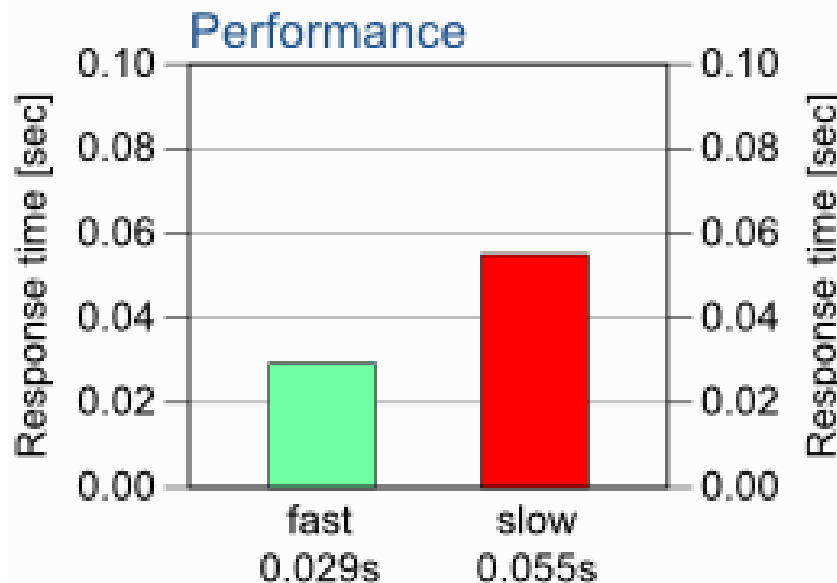
- Section are rolul de a controla volumul de date. Cu cat este mai mare section, cu atat este mai mare volumul de date returnat.
- Consideram doi indecsi: **index1** si **index2**

Volumul de date

- Interogarea analizata:

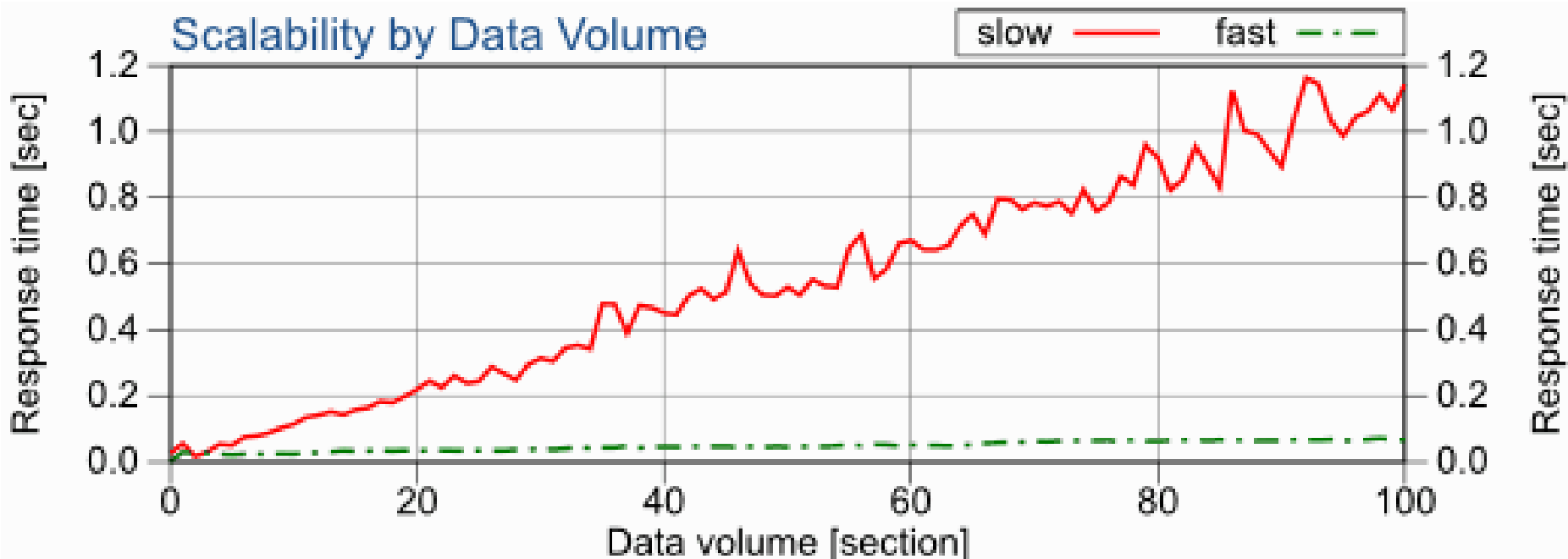
```
SELECT count(*) FROM scale_data  
WHERE section = ? AND id2 = ?
```

- *Section mic* – **index1** si apoi **index2**



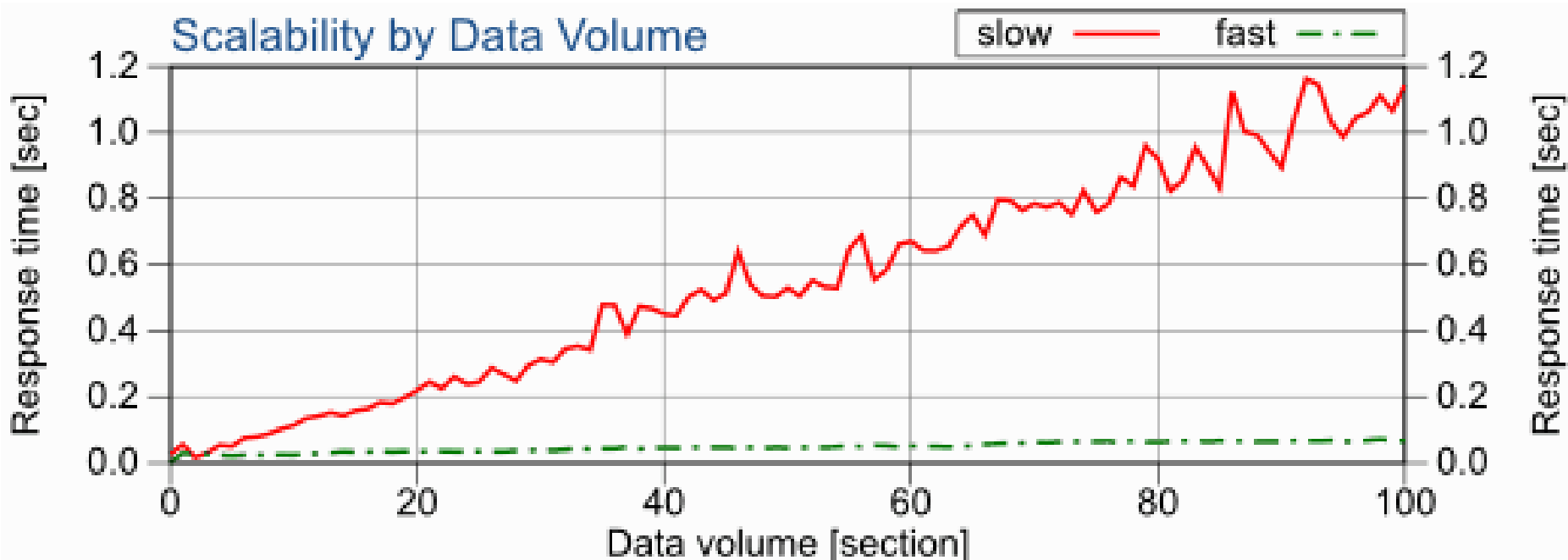
Volumul de date

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



Volumul de date

- **index1** – timp dublu fata de cel initial
- **index2** – timp 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

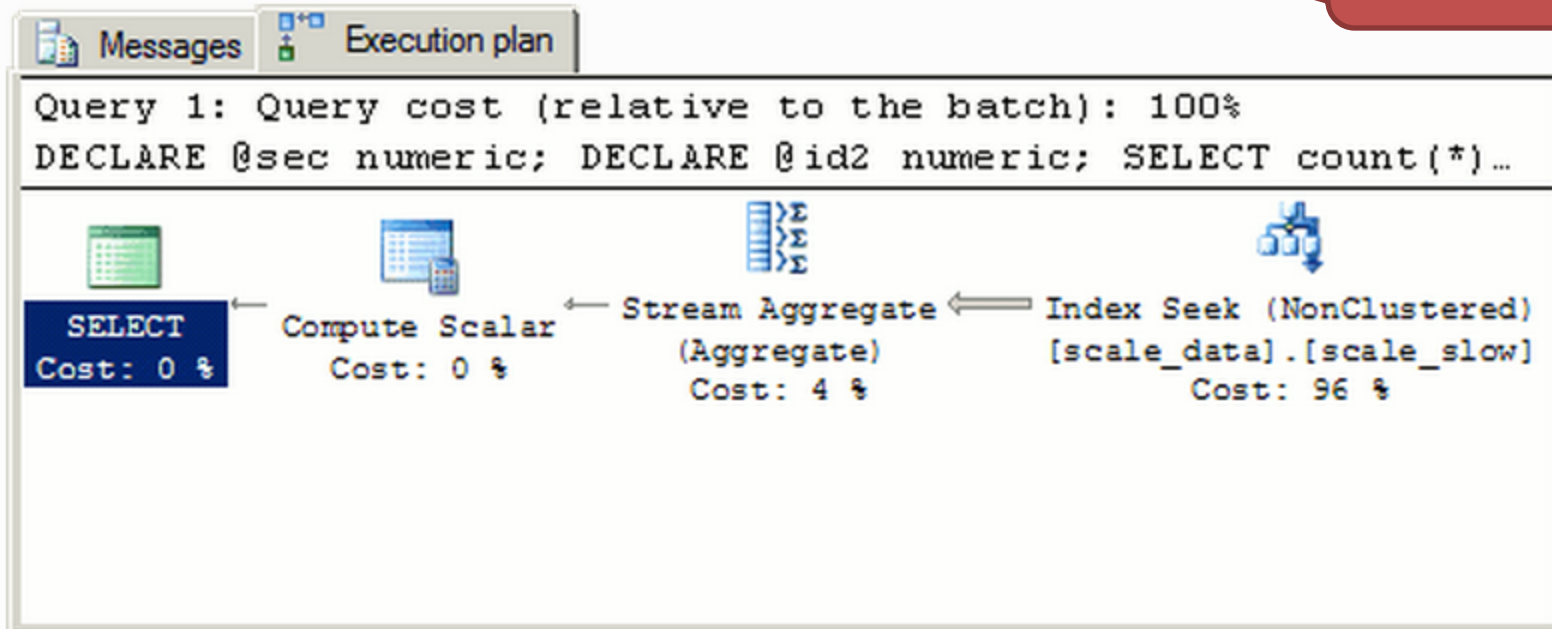
- Puteti spune cum a fost construit indexul avand execution plans ?
- **CREATE INDEX scale_slow ON scale_data (section, id1, id2);**
- **CREATE INDEX scale_fast ON scale_data (section, id2, id1);**

Campul id1 este adaugat doar pentru a pastra aceeaasi 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 interogariile noastre nu il face sa fie neaparat folosit de QO.
- *SQL Server Management Studio*

Arata predicatul doar ca un tooltip

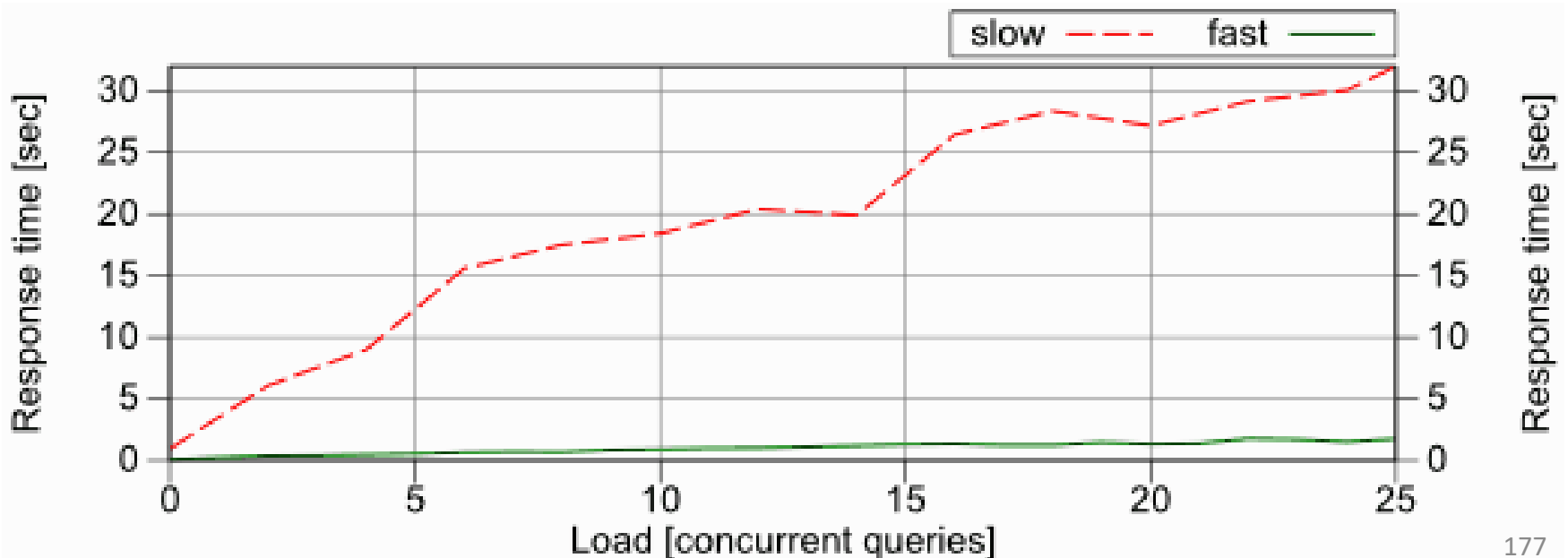


Incarcarea sistemului

- 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 *benchmarkuri* superficiale (gen *SQL Server Management Studio*).

Incarcarea sistemului

- Ne-am putea aștepta ca **hardwareul mai puternic din producție să ducă mai bine sistemul**. În fapt, în faza de development nu există deloc **latenta** – ceea ce nu se întâmplă în producție (**unde accesul poate fi întârziat din cauza rețelei**).
- <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 servește unui anumit scop.
- Sensibil la latente ale discului (și fragmentare).

Join

- Reducerea timpilor = indexarea corecta 😊
- Toti algoritmi de join proceseaza doar doua tabele simultan (apoi rezultatul cu a treia, etc).
- Rezultatele de la un join sunt trimise 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) peste UPPER(last_name)
 2. Un Index concatenat peste subsidiary_id, employee_id

Join – nested selects

- Totusi, in BD nu avem latentă din rețea.
- 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).

There you go: PLSQL style ;)

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 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)

| Applies to |
|------------|
| MySQL      |
| Oracle     |
| PostgreSQL |
| SQL Server |

# 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)

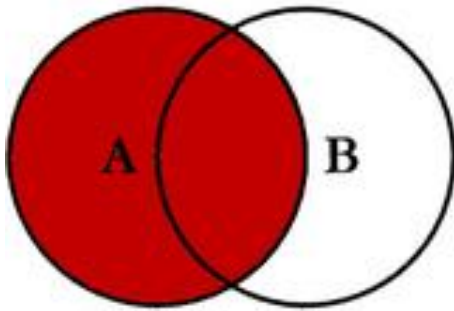
# Join – Hash join

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

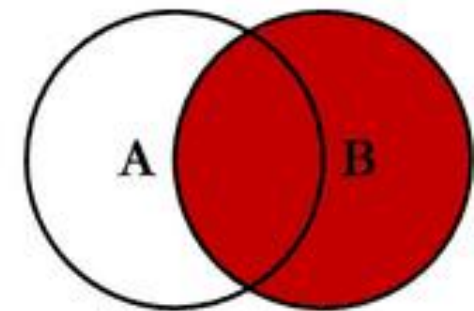
# Bibliografie (online)

- <http://use-the-index-luke.com/>  
( puteti cumpara si cartea in format PDF – dar nu contine altceva decat ceea ce este pe site)

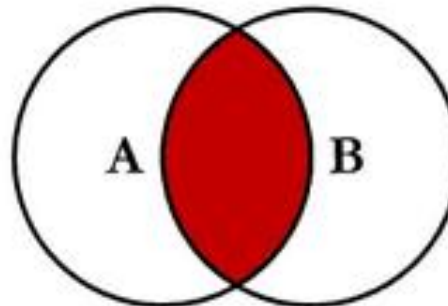
# SQL JOINS



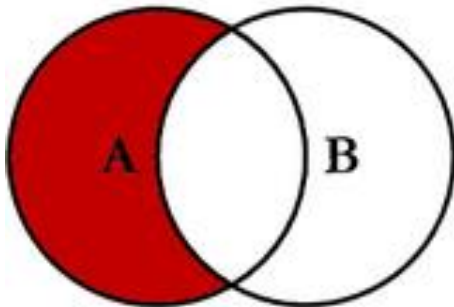
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



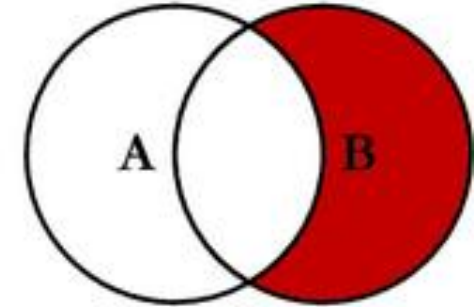
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



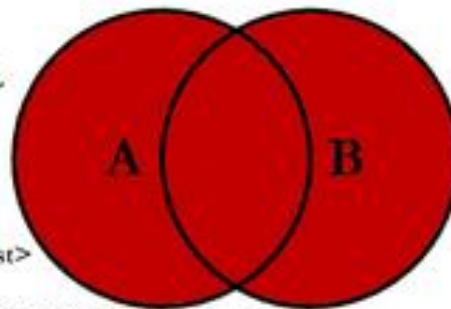
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



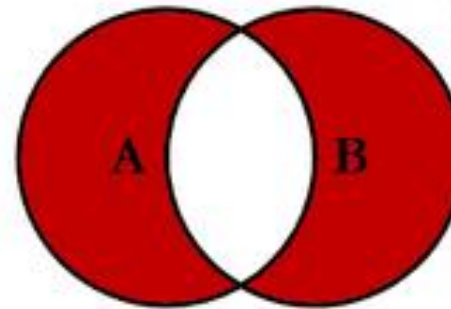
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
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
```