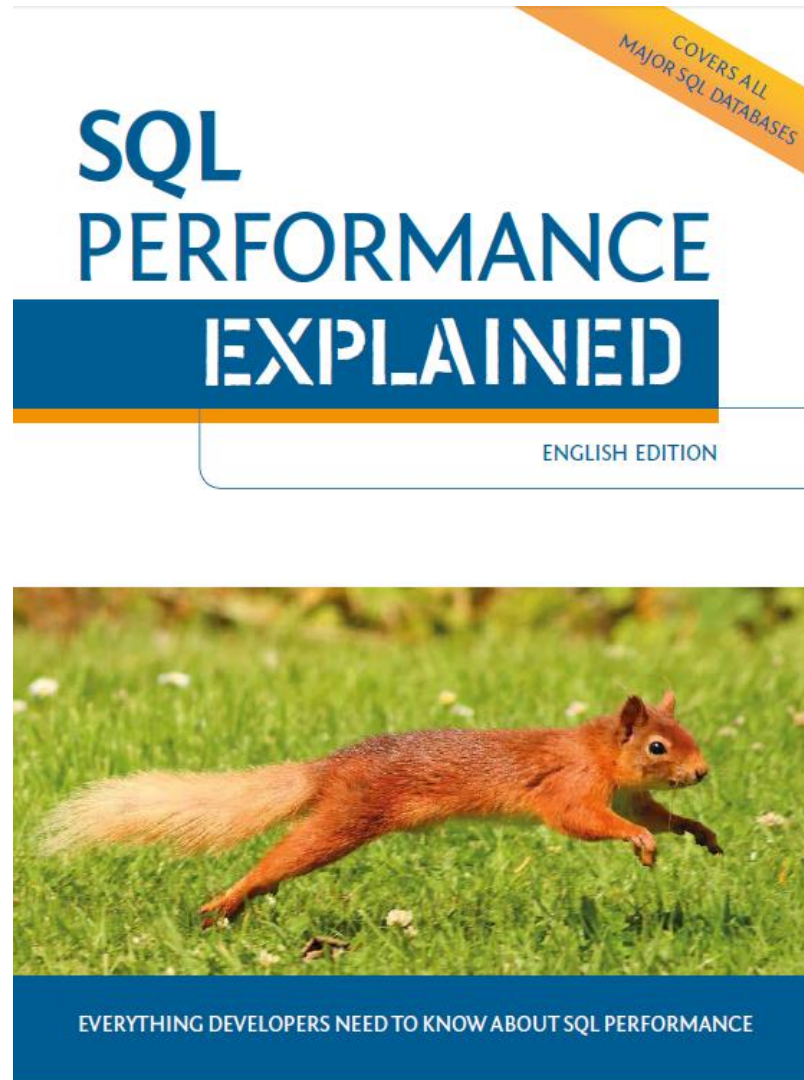


Practica SGBD

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



MARKUS WINAND

Join

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-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 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 = $n!$
- Nu este o problema cand sunt utilizati parametri dinamici [De ce ?]
- Nested Loops
- Hash Joins
- Merged Joins

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 latentă din rețea.
- Totusi, in BD nu sunt transferate datele intermediare (care sunt *pipelined* 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 hash pentru inregistrarile candidat.
- La *equijoin* in care una din tabele este foarte mare si cealalta este incarcata in memorie...
- *Hash join* imbunatatit daca sunt selectate mai putine coloane.
- A se indexa predicatele independente din where pentru a imbunatati eficienta. (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 eficienta 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)

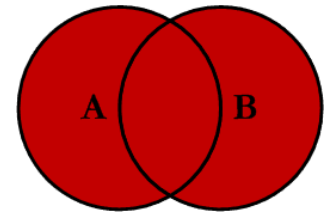
# Join – Hash join

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

# Join – Sort Merge

- Combina doua tabele preluand date fie dintr-unul, fie din celalalt, fie din ambele.
- Aceiasi indecsi ca si la hash-join (adica doar pentru conditiile separate, nu si pentru cele din join).
- Ordinea joinurilor nu conteaza.
- Algoritm foarte util pentru outer joins (in care sunt intoarse inregistrarile care fac match in ambele tabele) dar si pt L/R joins.

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



# Join – Sort Merge

|        |    |
|--------|----|
| Smith  | 40 |
| Winand | 44 |
| Tyler  | 46 |

|    |         |
|----|---------|
| 40 | EUR 100 |
| 40 | EUR 250 |
| 42 | EUR 100 |
| 44 | EUR 500 |

FULL OUTER JOIN using sort-merge



# Join – Sort Merge

- Desul de greoi de utilizat din cauza ca ambele tabele trebuie sa fie sortate dupa campul utilizat in join.



# Clustering Data

# Clustering Data

- Un cluster = serie de obiecte (de obicei de aceeași natură) ce apar împreună.
- Un cluster de calculatoare este o grupare de calculatoare care “se ajută” simultan.
- Utilizate pentru
  - a rezolva o problemă complexă (high performance clusters)
  - a mări disponibilitatea (failover cluster)

# Clustering Data

- In cazul calculatoarelor mai exista un tip de cluster: **Data Cluster**
- Clustering data = asezarea datelor ce au probabilitatea imediata de acces unele langa altele (de exemplu atunci cand defragmentam HDD-ul, facem acest tip de clusterizare).
- Computer cluster este destul de intalnit si intre cele doua se face confuzie (“Let’s use a cluster to improve DB performance”)

# Clustering Data

1. Cel mai simplu tip de cluster este randul: bazele de date **stocheaza toate coloanele dintr-un rand in acelasi block** (DB block) daca e posibil.

## *Column Stores*

*Column oriented databases, or column-stores, organize tables in a columned way. This model is beneficial when accessing many rows but only a few columns—a pattern that is very common in data warehouses (OLAP).*

# Clustering Data

2. Indecsii construiesc clustere de randuri – frunzele B-tree-ului **stocheaza coloanele intr-o maniera ordonata** (valorile consecutive sunt puse unele langa altele).
- Indecsii construiesc clustere de randuri cu valori similare = *second power of indexing*.

# Clustering Data- Index Filter Predicates Used Intentionally

- Cand apare “*Index filter*” de multe ori inseamna ca indexul nu a fost construit corect si ca probabil exista o varianta mai eficienta (in care ceea ce este filtrat este folosit de index).
- *Index filtering* poate fi folosit si intr-un scop bun: pentru gruparea datelor ce sunt accesate consecutiv.

# Clustering Data- Index Filter Predicates Used Intentionally

- Clauzele WHERE ce nu pot fi utilizate ca si predicate de acces sunt cele mai bune pentru aceasta tehnica:

```
SELECT first_name, last_name, subsidiary_id
FROM employees
WHERE subsidiary_id = ?
 AND UPPER(last_name) LIKE '%INA%';
```

Indexul e inutil indiferent daca e dupa **last\_name** sau dupa **upper(last\_name)**. Dupa ce indexam ?



# Clustering Data- Index Filter Predicates Used Intentionally

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 17   | 230  |
| *1 | TABLE ACCESS BY INDEX ROWID | EMPLOYEES   | 17   | 230  |
| *2 | INDEX RANGE SCAN            | EMPLOYEE_PK | 333  | 2    |

Predicate Information (identified by operation id):

- 1 - filter(UPPER("LAST\_NAME") LIKE '%INA%')
- 2 - access("SUBSIDIARY\_ID"=TO\_NUMBER(:A))

- Indexare dupa SUBSIDIARY\_ID

# Clustering Data- Index Filter Predicates Used Intentionally

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 17   | 230  |
| *1 | TABLE ACCESS BY INDEX ROWID | EMPLOYEES   | 17   | 230  |
| *2 | INDEX RANGE SCAN            | EMPLOYEE_PK | 333  | 2    |

Predicate Information (identified by operation id):

- 1 - filter(UPPER("LAST\_NAME") LIKE '%INA%')
- 2 - access("SUBSIDIARY\_ID"=TO\_NUMBER(:A))

- Indexare dupa SUBSIDIARY\_ID

# Clustering Data- Index Filter Predicates Used Intentionally

- *Table acces* nu este neaparat o povara daca inregistrările ar fi stocate grupat in acelasi block si BD ar putea sa le returneze intr-o singura operatie de tip *read*.
- Daca randurile sunt dispersate, BD va trebui sa citeasca din fiecare block in parte.
- Eficienta depinde de distributia fizica a datelor.

# Clustering Data- Index Filter Predicates Used Intentionally

- Corelatia dintre ordinea din index si cea din tabela se numeste ***index clustering factor***.  
(nr blocuri < ICF < nr randuri)
- Este posibila imbunatatirea eficientei re-aranjand randurile din tabel (ce se intampla daca avem doi indecsi diferiti ?)
- Chiar daca am face ordonarea dupa un singur index, si asa e complicat pentru ca BD nu permite decat un acces rudimentar la aceasta ordine (*row sequencing*).

# Clustering Data- Index Filter Predicates Used Intentionally

- Aici apare partea cu clusterizarea: utilizarea unui index din doua coloane pentru a fi stocate intr-o ordine bine definita.
- Se va extinde indexul pentru a acoperi chiar si coloana “neindexabila”:

```
CREATE INDEX empsubupnam ON employees
 (subsidiary_id, UPPER(last_name));
```

# Clustering Data- Index Filter Predicates Used Intentionally

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 17   | 20   |
| 1  | TABLE ACCESS BY INDEX ROWID | EMPLOYEES   | 17   | 20   |
| *2 | INDEX RANGE SCAN            | EMPSUBUPNAM | 17   | 3    |

Predicate Information (identified by operation id):

```
2 - access("SUBSIDIARY_ID"=TO_NUMBER(:A))
 filter(UPPER("LAST_NAME") LIKE '%INA%')
```

- *Execution plan* e identic dar costul este mult mai mic pentru table access. Se observa ca filtrul este direct aplicat in pasul 2, pasul 2 returneaza doar 17 randuri (in cazul anterior erau 333).

```
SELECT first_name, last_name, subsidiary_id, phone_number
FROM employees
WHERE SUBSIDIARY_ID = 30
AND UPPER(last_name) LIKE '%INA%'
```

---

## Intentional Index Filter-Predicates for LIKE

```
SELECT first_name, last_name, subsidiary_id, phone_number
FROM employees
WHERE SUBSIDIARY_ID = 30
AND UPPER(last_name) LIKE '%INA%'
```

# Clustering Data- Index Filter Predicates Used Intentionally

- Aceasta abordare ignora relevanta ordinii coloanelor. De obicei, partea ce va fi filtrata este bine sa fie pusa ultima (pentru ca indexul sa mearga corect pe primele coloane).
- Nu adaugati toate coloanele in index.



# Clustering Data- Index-Only Scan

- Una dintre cele mai puternice metode de *“tuning”*.
- Nu numai ca evita evaluarea clauzei *where* dar evita accesarea tabelului atunci cand coloanele selectate se gasesc chiar in index.

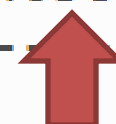
# Clustering Data- Index-Only Scan

```
CREATE INDEX sales_sub_eur
 ON sales (subsidiary_id, eur_value);

SELECT SUM(eur_value)
 FROM sales
 WHERE subsidiary_id = ?;
```

# Clustering Data- Index-Only Scan

| Id  | Operation        | Name          | Rows  | Cost |
|-----|------------------|---------------|-------|------|
| 0   | SELECT STATEMENT |               | 1     | 104  |
| 1   | SORT AGGREGATE   |               | 1     |      |
| * 2 | INDEX RANGE SCAN | SALES_SUB_EUR | 40388 | 104  |



Predicate Information (identified by operation id):

2 - access("SUBSIDIARY\_ID"=TO\_NUMBER(:A))

# Clustering Data- Index-Only Scan

- Indexul acopera intreaga interogare si din acest motiv se mai numeste “**covering index**” (poate fi identificat prin faptul ca nu este accesata deloc tabela ci doar indexul).
- Numarul mare de linii selectate poate avea un impact negativ asupra eficientei (mai ales daca nu sunt in acelasi cluster – in ex, s-ar putea ca eur\_value sa nu fie a doua coloana din index (ci a treia) ceea ce ar face ca valorile lui sa fie in clustere diferite).

# Clustering Data- Index-Only Scan

- Index-only este o strategie agresiva.
- **Nu proiectati indexul in acest scop** pentru ca e ca si cum ati copia tabela cu bucatile care va intereseaza – ocupa spatiu si timp pentru operatii *insert/edit/delete*.

# Clustering Data- Index-Only Scan

- Index-only scan poate cauza surprize neplacute

```
SELECT SUM(eur_value)
 FROM sales
 WHERE subsidiary_id = ?
 AND sale_date > ?;
```

- Ce se intampla ?

# Clustering Data- Index-Only Scan

- Scrieti comentarii cand folositi selecturi de tipul Index-Only scan pentru a va asigura ca nu adaugati din greseala noi predicate ce vor impiedica accesul in stilul index-only scan.

# Clustering Data- Index-Only Scan

- Alta problema poate apare de la FBI.
- UPPER(LAST\_NAME) nu poate fi folosit in Index Only Scan daca se doreste selectarea coloanei LAST\_NAME (pentru ca indexul tine minte valoarea functiei si nu valoarea coloanei) – incercati sa indexati valoarea originala daca stiti ca o veti folosi tot pe ea.

Homework: [Testati viceversa]



# Clustering Data- Index-Only Scan

- Mereu selectati doar coloanele de care aveti nevoie. Daca selectati de genul “select \* ...” in nici un caz nu veti folosi Index-Only Scan.
- Exista anumite limitari in ceea ce priveste dimensiunea indexului (deci nu puteti pune tot in index pentru a face numai Index-only Scan):

# Limitations of Index-Only Scan

- The maximum index key length depends on the block size and the index storage parameters (75% of the database block size minus some overhead). A B-tree index is limited to 32 columns.
- When using Oracle 11g with all defaults in place (8k blocks), the maximum index key length is 6398 bytes. Exceeding this limit causes the error message “ORA-01450: maximum key length (6398) exceeded.”

# Clustering Data- Index-Only Scan

- Interogările care nu selectează nici o coloană din tabel sunt executate cu Index-Only scans.

Puteti gasi un exemplu ?

# Clustering: Index-Organized Tables

- Index-Only Scan executa interogarea SQL doar folosind informatiile redundante existente in index.
- Daca am pune toate informatiile in index, de ce am mai avea nevoie de tabela originala ?
- Intr-adevar, exista aceasta abordare: *index-organized tables* (IOT) sau *clustered index*.

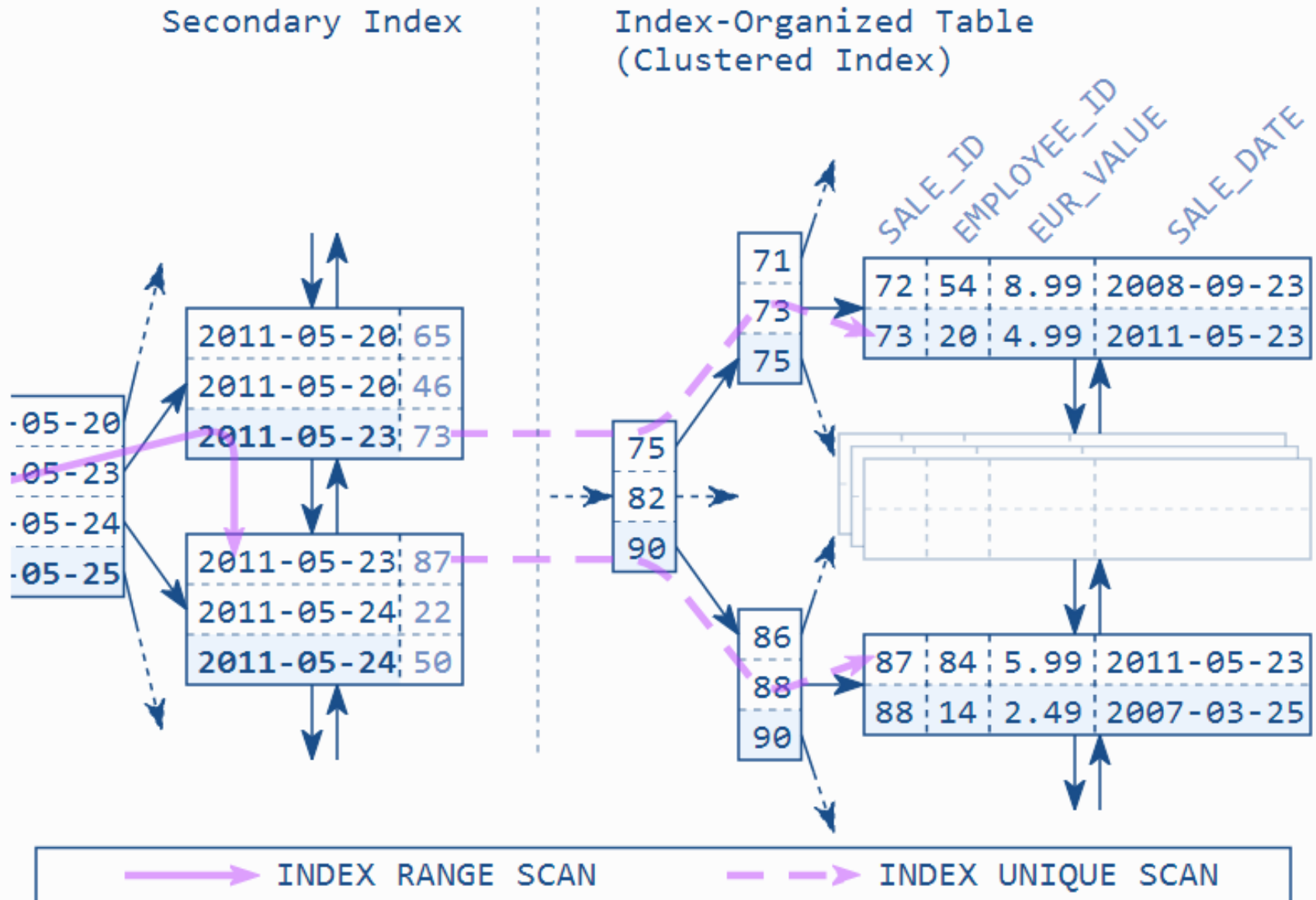
# Clustering: Index-Organized Tables

- Avantaje:
  - nu se mai foloseste tabelul si se poate renunta la el;
  - orice acces intr-un IOT este un index-only scan.
- Ambele suna bine dar sunt greu de obtinut in practica.

# Clustering: Index-Organized Tables

- Problema: cand se doreste un nou index pe aceeasi tabela: noul index va pointa prin intermediul unei chei catre informatia logica din indexul original (si va face index unique scan dupa aceasta cheie).
- Accesarea unui IOT dupa un al doilea index este foarte ineficienta.

# Clustering: Index-Organized Tables



# Clustering: Index-Organized Tables

- Al doilea index pastreaza clustering keys pentru fiecare inregistrare.
- Am putea de exemplu sa interogam doar ID-urile (si accesul ar index-scan only):



# Clustering: Index-Organized Tables

```
SELECT sale_id
FROM sales_iot
WHERE sale_date = ?;
```

| ----- |                  |                |      |  |
|-------|------------------|----------------|------|--|
| Id    | Operation        | Name           | Cost |  |
| ----- |                  |                |      |  |
| 0     | SELECT STATEMENT |                | 4    |  |
| * 1   | INDEX RANGE SCAN | SALES_IOT_DATE | 4    |  |
| ----- |                  |                |      |  |

Predicate Information (identified by operation id):

-----  
1 - access("SALE\_DATE"=:DT)

# Clustering: Index-Organized Tables

- Dar daca ne intereseaza altceva decat ID-ul (care cel mai probabil nu e interesant), accesul va trebui facut si in celalalt index (cel original)

# Clustering: Index-Organized Tables

```
SELECT eur_value
FROM sales_iot
WHERE sale_date = ?;
```

| ----- |  |                   |                |      |
|-------|--|-------------------|----------------|------|
| Id    |  | Operation         | Name           | Cost |
| ----- |  |                   |                |      |
| 0     |  | SELECT STATEMENT  |                | 13   |
| * 1   |  | INDEX UNIQUE SCAN | SALES_IOT_PK   | 13   |
| * 2   |  | INDEX RANGE SCAN  | SALES_IOT_DATE | 4    |
| ----- |  |                   |                |      |

Predicate Information (identified by operation id):

- 
- 1 - access("SALE\_DATE"=:DT)
  - 2 - access("SALE\_DATE"=:DT)

# Clustering: Index-Organized Tables

- Eficienta este pierduta atunci cand este folosit un al doilea index.
- Clustering key este de obicei mai lung decat ROWID (deci si indecsii secundari vor ocupa mai mult spatiu).
- Concluzia: tabelele ce folosesc un singur index sunt cele mai bine sa fie implementate ca un IOT.

# Clustering: Index-Organized Tables

```
CREATE TABLE (
 id NUMBER NOT NULL PRIMARY KEY,
 [...]
) ORGANIZATION INDEX;
```



Indecsii in sortare  
(ORDER BY)

# Sortarea

- Sortarea este o actiune anevoioasa pentru bazele de date, in principal pentru ca toate inregistrarile trebuie aduse intr-un buffer in care sa se faca sortarea.
- **Indecsii retin informatiile intr-o maniera ordonata** – deci putem utiliza un index pentru a usura munca operatiilor de tip “*order by*”.

# Sortarea

- Un INDEX RANGE SCAN care intoarce multe randuri poate fi ineficient – pentru ca timpul necesar pentru row acces ar putea fi mai mare decat daca s-ar face FULL TABLE SCAN si apoi s-ar sorta rezultatele.



# Sortarea

- Un index construit dupa criteriul ORDER BY va evita operatia de sortare si va putea intoarce foarte rapid inregistrarile dintr-un anumit interval.
- Indexul face ca operatiile de tip ORDER BY sa fie executate intr-o maniera pipelined = **3<sup>rd</sup> power of index**.
- *Pipelining negates the need to build huge collections by piping rows out of the function as they are created, saving memory and allowing subsequent processing to start before all the rows are generated.*
- Din cauza ca sunt gata sortate in index, nu trebuie sa le mai preia din tabela si sa le sorteze. Poate sa citeasca primul rowid direct din index si sa stie ca e primul, sa se duca apoi in tabela si sa preia restul informatiilor si sa le faca emit apoi sa mearga pe al doilea rowid etc....



# Indexarea pentru ORDER BY

- Interogările SQL cu o clauza ORDER BY nu trebuie sa sorteze rezultatul daca indexul deja ofera aceasta ordine:

```
SELECT sale_date, product_id, quantity
FROM sales
WHERE sale_date = TRUNC(sysdate) -
 INTERVAL '1' DAY
ORDER BY sale_date, product_id;
```

de fapt, e aceeași zi... deci ordonarea va fi făcută după product\_id...

# Indexarea pentru ORDER BY


- Exista un index dupa **sale\_date** totusi, va trebui facuta o operatie de sortare pentru a satisface clauza ORDER BY (product\_id):

| Id | Operation                   | Name       | Rows | Cost |  |
|----|-----------------------------|------------|------|------|--|
| 0  | SELECT STATEMENT            |            | 320  | 18   |  |
| 1  | SORT ORDER BY               |            | 320  | 18   |  |
| 2  | TABLE ACCESS BY INDEX ROWID | SALES      | 320  | 17   |  |
| *3 | INDEX RANGE SCAN            | SALES_DATE | 320  | 3    |  |

# Indexarea pentru ORDER BY

- Pentru a scapa de sortare, ar trebui ca indexul sa fie construit peste ambele campuri:

```
DROP INDEX sales_date;
CREATE INDEX sales_dt_pr ON sales
 (sale_date, product_id);
```



| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 320  | 300  |
| 1  | TABLE ACCESS BY INDEX ROWID | SALES       | 320  | 300  |
| *2 | INDEX RANGE SCAN            | SALES_DT_PR | 320  | 4    |

# Indexarea pentru ORDER BY

- Chiar daca numarul operatiilor e mai mic [nu se mai face SORT ORDER BY], costul a crescut (din cauza ca datele **nu** sunt “clusterizate”).
- Clusterizarea pentru indexul ce utiliza doar sale\_date este la minim, indexul fiind construit din sale\_date si pentru ordinea produselor din aceeasi zi (pentru ordonare) este utilizat ROWID. La adaugarea product\_id, nu mai are aceeasi libertate de ordonare a informatiilor...
- Probabil, produsele vandute in aceeasi zi sunt pastrate in tabela consecutiv (dar nu si in ordinea IDurilor).

# Indexarea pentru ORDER BY

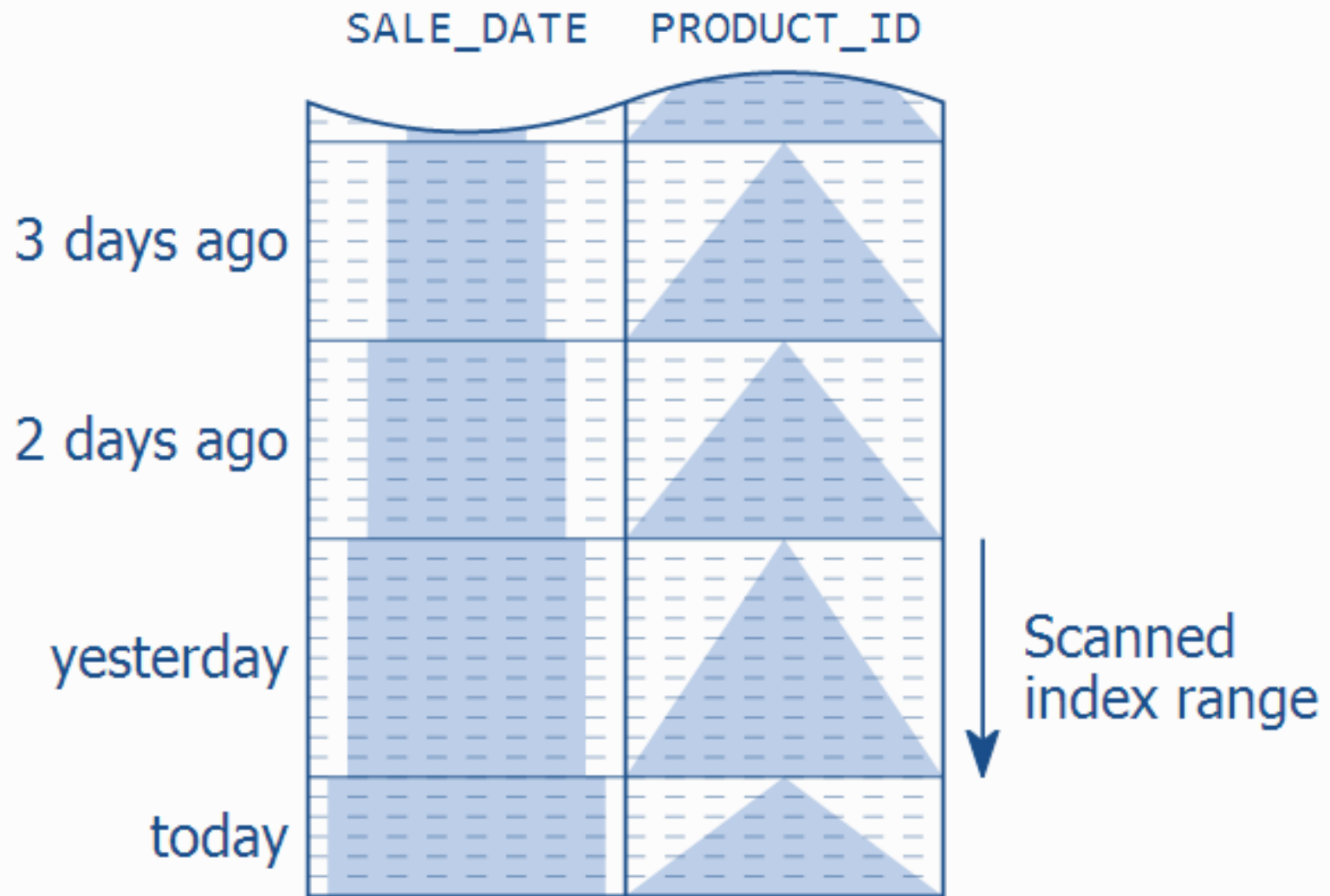
- Adica...
- Daca toate vanzarile de ieri se afla in doua blocuri, ele vor fi preluate toate din doua citiri dupa care vor fi ordonate.
- Daca sunt preluate din index s-ar putea ca, desi sunt direct ordonate, pentru fiecare rand sa se faca cate o citire din bloc1, bloc2, bloc1, bloc2 etc....

# Indexarea pentru ORDER BY

- Cum produsele cerute sunt oricum din aceeași zi, e suficient să facem ordonarea doar după `product_id` (ca să nu citească de fiecare dată – din nou valoarea lui `sale_date` – oricum toate sunt vândute în aceeași zi [WHERE...]).

```
SELECT sale_date, product_id, quantity
FROM sales
WHERE sale_date = TRUNC(sysdate) -
 INTERVAL '1' DAY
ORDER BY product_id;
```

# Indexarea pentru ORDER BY





# Indexarea pentru ORDER BY

- Aceasta optimizare poate fi neplacuta daca marim intervalul cautat:

```
SELECT sale_date, product_id, quantity
FROM sales
WHERE sale_date >= TRUNC(sysdate) -
 INTERVAL '1' DAY
ORDER BY product_id;
```

# Indexarea pentru ORDER BY

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 320  | 301  |
| 1  | <b>SORT ORDER BY</b>        |             | 320  | 301  |
| 2  | TABLE ACCESS BY INDEX ROWID | SALES       | 320  | 300  |
| *3 | INDEX RANGE SCAN            | SALES_DT_PR | 320  | 4    |

- Se observa ca apare iarasi “SORT ORDER BY”

# Indexarea pentru ORDER BY

- Daca BD foloseste o operatie de sortare chiar cand va asteptati la o executie *pipelined*, motivul este unul din urmatoarele doua:
  - QO considera ca executia este optima in acest fel.
  - Ordinea indecsilor nu corespunde clauzei “ORDER BY”
- Puteti sa va dati seama de motiv folosind un *full index definition* in clauza *order by*.

# Indexarea pentru ORDER BY

- Atunci cand totusi QO prefera sa sorteze lista, acest lucru se intampla pentru ca prefera sa faca operatia peste toata lista (de exemplu pentru ca oricum trebuie sa incarce toate inregistrarile) si ordonarea nu i-ar lua mai mult timp decat daca ar citi din indecsi si ar accesa tabela pentru fiecare rowid.

# ASC/DESC/NULL FIRST, LAST

- BD pot citi indecsii in ambele directii.
- Indecsii sunt folositi si daca ORDER BY trebuie sa intoarca rezultatul in ordine inversa.
- Modificatorii ASC/DESC **pot** schimba executia de tip pipedline intr-una in care trebuie facuta sortarea.

| Applies to      |
|-----------------|
| DB2<br>✓        |
| MySQL<br>✗      |
| Oracle<br>✓     |
| PostgreSQL<br>✓ |
| SQL Server<br>✓ |

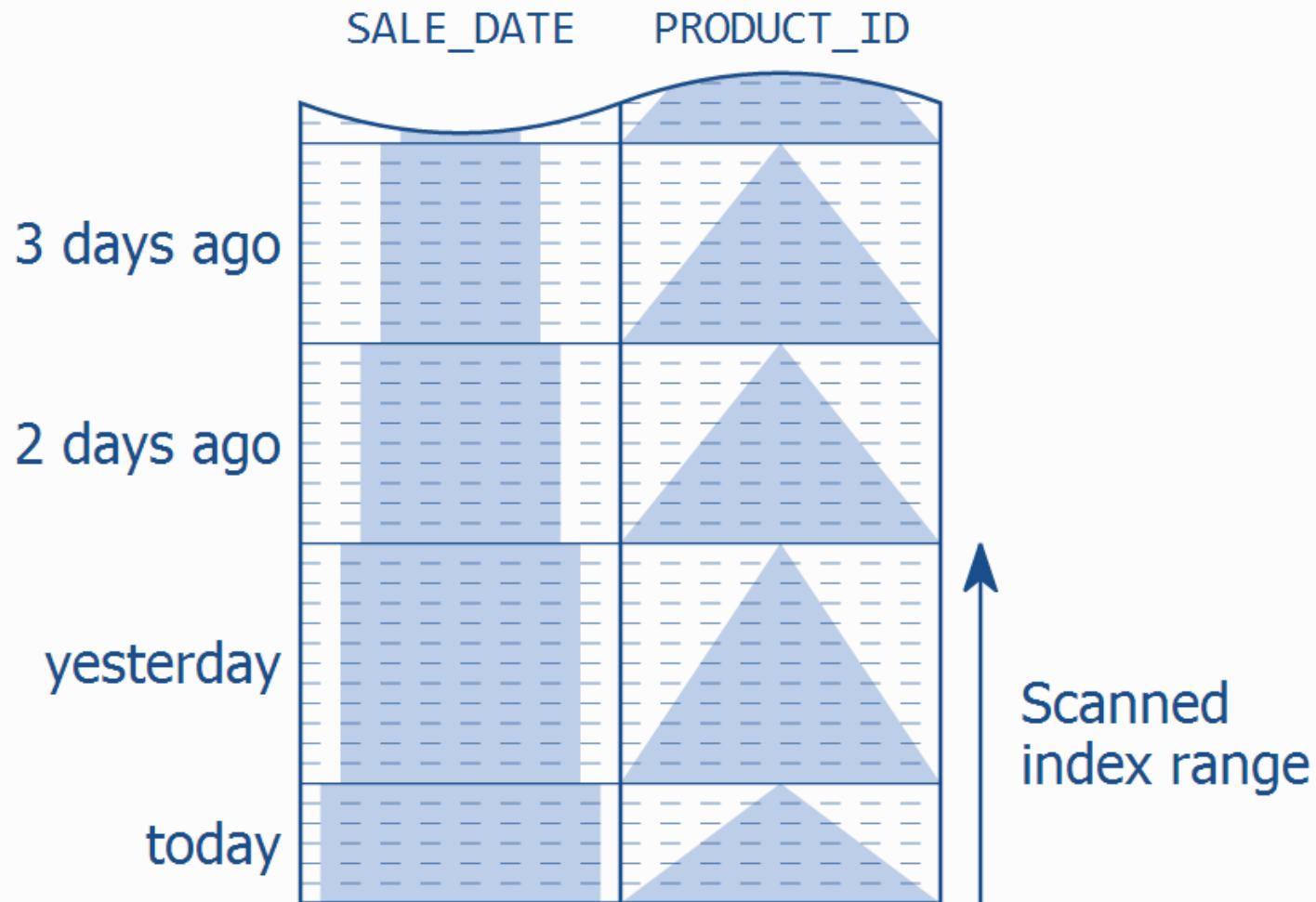
# ASC/DESC/NULL FIRST, LAST

```
SELECT sale_date, product_id, quantity
FROM sales
WHERE sale_date >= TRUNC(sysdate) -
 INTERVAL '1' DAY
ORDER BY sale_date DESC, product_id DESC;
```

- In acest caz, planul de executie arata astfel:

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 320  | 300  |
| 1  | TABLE ACCESS BY INDEX ROWID | SALES       | 320  | 300  |
| *2 | INDEX RANGE SCAN DESCENDING | SALES_DT_PR | 320  | 4    |

# ASC/DESC/NULL FIRST, LAST



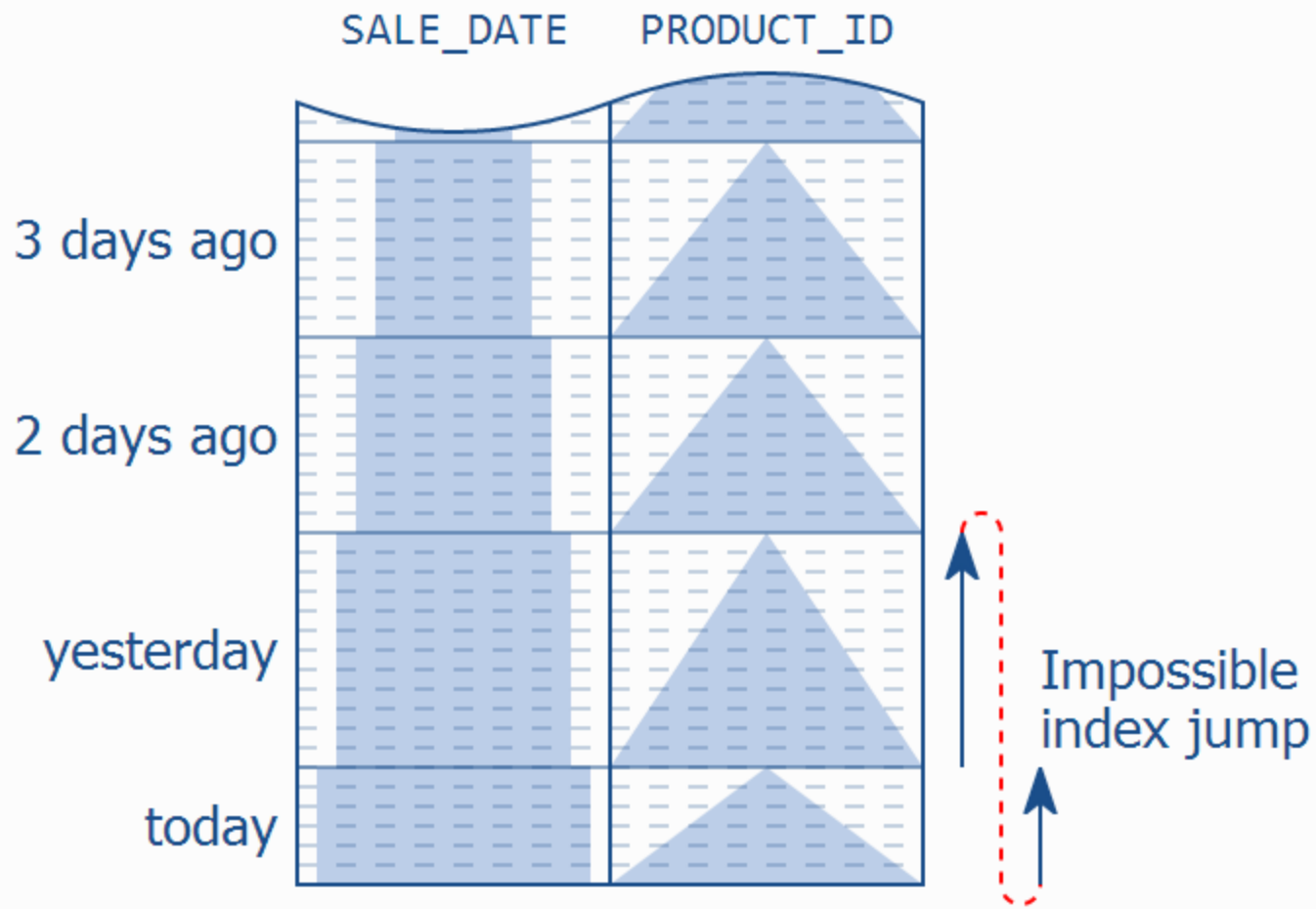
# ASC/DESC/NULL FIRST, LAST

- Este crucial ca indexul sa fie EXACT in ordine inversa fata de ce se cere in clauza ORDER BY!

```
SELECT sale_date, product_id, quantity
FROM sales
WHERE sale_date >= TRUNC(sysdate) -
INTERVAL '1' DAY
ORDER BY sale_date ASC, product_id DESC;
```



# ASC/DESC/NULL FIRST, LAST



# ASC/DESC/NULL FIRST, LAST

- BD nu va putea folosi indexul pentru a evita sortarea (va prelua din ziua precedenta si va sorta apoi dupa product\_id apoi din ziua curenta si va sorta iarasi dupa product\_id).
- Pentru cazuri ca acesta, cele mai multe BD ofera posibilitatea de a crea indecsi intr-o anumita ordine.

# ASC/DESC/NULL FIRST, LAST

```
DROP INDEX sales_dt_pr;
```

```
CREATE INDEX sales_dt_pr
ON sales (sale_date ASC, product_id DESC);
```

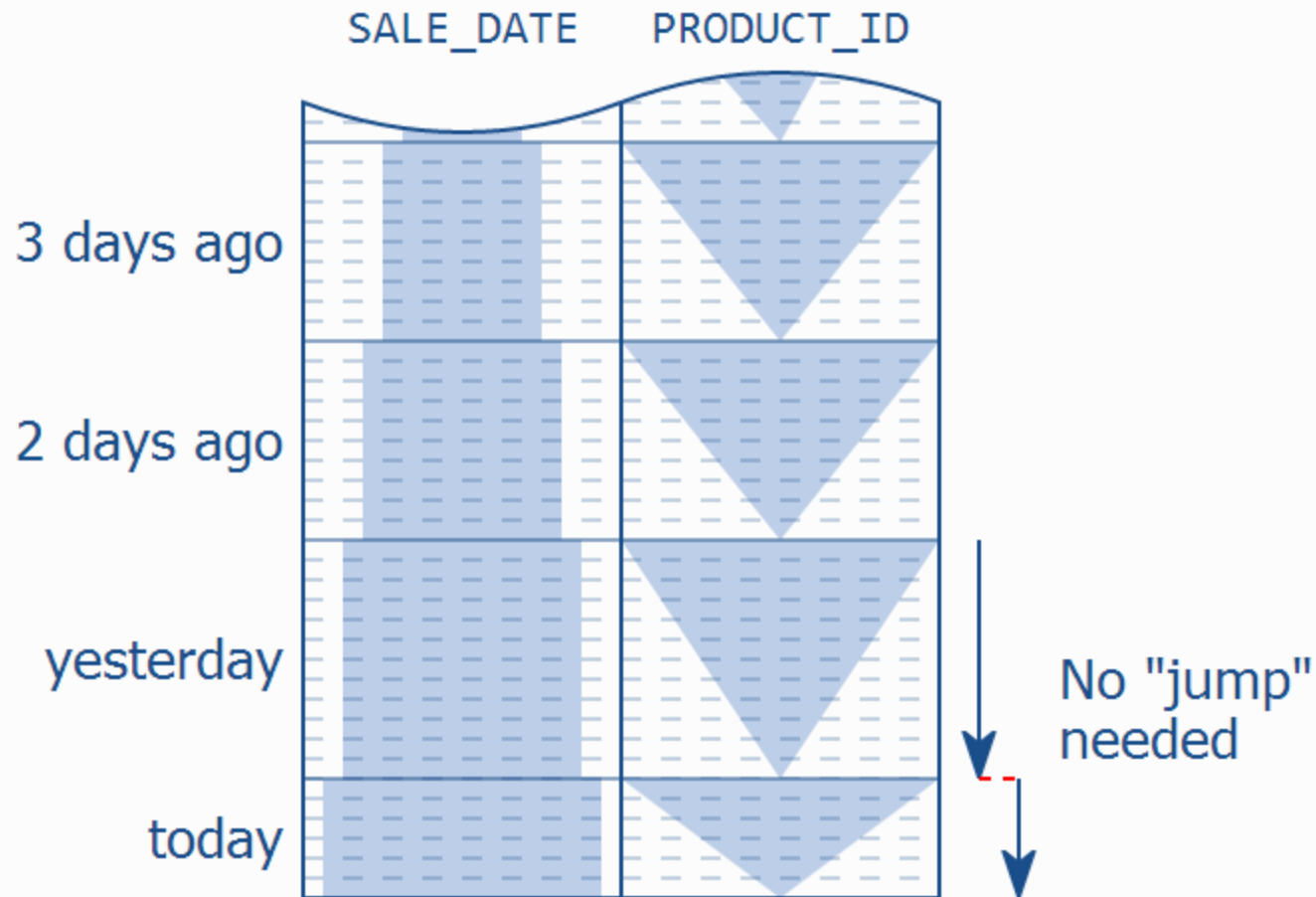
- [MySQL ignora ASC/DESC din definitia indecsilor]
- Noua definitie nu afecteaza cautarea in clauza WHERE.

# ASC/DESC/NULL FIRST, LAST

- Dupa crearea noului index, se va evita sortarea (din nou):

| Id | Operation                   | Name        | Rows | Cost |  |
|----|-----------------------------|-------------|------|------|--|
| 0  | SELECT STATEMENT            |             | 320  | 301  |  |
| 1  | TABLE ACCESS BY INDEX ROWID | SALES       | 320  | 301  |  |
| *2 | INDEX RANGE SCAN            | SALES_DT_PR | 320  | 4    |  |

# ASC/DESC/NULL FIRST, LAST



# ASC/DESC/NULL FIRST, LAST

- ASC/DESC sunt utilizate pentru a inversa ordinea anumitor coloane. Nu este nevoie sa le sortam pe toate DESC. De ce ?

# ASC/DESC/NULL FIRST, LAST

- Inafara de ASC/DESC, SQL permite inca doi modifcatori pentru ORDER BY (NULL FIRST sau NULL LAST).
- Problema apare din cauza ca nu se poate specifica ordinea pentru NULL (in index).
- Nu este implementata nici de SQL server 2012 si nici de MySQL 5.6.
- Oracle SQL suporta sortarea NULL-urilor inainte sa fie introdus in standard dar nu permite indexarea lor nici macar in *11g*.

# ASC/DESC/NULL FIRST, LAST

- Oracle nu va permite *pipeline* atunci cand se va face sortare cu NULLS FIRST.
- PostgreSQL suporta (din ver 8.3) modificatori NULLS atat in ORDER BY cat si in definirea indecsilor.



# ASC/DESC/NULL FIRST, LAST

|                           | DB2   | MySQL | Oracle | PostgreSQL | SQLite | SQL Server |
|---------------------------|-------|-------|--------|------------|--------|------------|
| Read index backwards      | ✓     | ✓     | ✓      | ✓          | ✓      | ✓          |
| Order by ASC/DESC         | ✓     | ✓     | ✓      | ✓          | ✓      | ✓          |
| Index ASC/DESC            | ✓     | ✗     | ✓      | ✓          | ✓      | ✓          |
| Order by NULLS FIRST/LAST | ✗     | ✗     | ✓      | ✓          | ✗      | ✗          |
| Default NULLS order       | First | First | Last   | Last       | First  | First      |
| Index NULLS FIRST/LAST    | ✗     | ✗     | ✗      | ✓          | ✗      | ✗          |



# Indecsii in grupare (GROUP BY)

# Indexarea GROUP BY

- BD SQL folosesc doi algoritmi diferiti:
  - 1) algoritm **Hash** care face agregare de inregistrari intr-o tabela hash. Dupa ce toate inputurile au fost procesate, se returneaza tabela hash (nu si in MySQL 5.6)
  - 2) **sort/group**: intai sorteaza datele dupa cheile de grupare a.i. ele sa fie consecutive dupa care le face agregare.

| Applies to   |
|--------------|
| DB2 ✓        |
| MySQL ✓      |
| Oracle ✓     |
| PostgreSQL ✓ |
| SQL Server ✓ |

# Indexarea GROUP BY

- Ambii algoritmi au pasi intermediari (construire de **hashuri** / **sortarea**) si din acest motiv nu pot fi executati in maniera *pipelined*.
- Totusi, algoritmul sort/group (al doilea) poate utiliza un index pentru a evita sortarea, in acest fel dand posibilitatea de efectuare a *pipeline* chiar si in cadrul gruparilor.

# Indexarea GROUP BY

- Exemplu:

```
SELECT product_id, sum(eur_value)
 FROM sales
 WHERE sale_date = TRUNC(sysdate) -
 INTERVAL '1' DAY
 GROUP BY product_id;
```

| Id | Operation                         | Rows | Cost |
|----|-----------------------------------|------|------|
| 0  | SELECT STATEMENT                  | 17   | 192  |
| 1  | SORT GROUP BY <b>NOSORT</b>       | 17   | 192  |
| 2  | TABLE ACCESS BY INDEX ROWID SALES | 321  | 192  |
| *3 | INDEX RANGE SCAN SALES_DT_PR      | 321  | 3    |

Pipelined GROUP BY  
(Oracle)

# Indexarea GROUP BY

- Pentru a se produce un *pipelined group by*, trebuie sa fie indeplinite aceleasi conditii ca si in cazul lui *order by*.
- Aceeasi observatie pentru NULLS FIRST/LAST.
- Exista baze de date care nu pot procesa corect ASC/DESC indexarea pentru a face *pipelined group by* [In PostgreSQL trebuie adaugat un *order by* pentru a face un NULLS LAST ce va fi folosit in *group by*].

# Indexarea GROUP BY

- In Oracle **nu** se poate citi un index in ordine inversa pentru a se executa un *piped group by* daca este urmat de un *order by*.

# Indexarea GROUP BY

```
SELECT product_id, sum(eur_value)
FROM sales
WHERE sale_date >= TRUNC(sysdate) -
 INTERVAL '1' DAY
GROUP BY product_id;
```

| Id | Operation                   | Name        | Rows | Cost |
|----|-----------------------------|-------------|------|------|
| 0  | SELECT STATEMENT            |             | 24   | 356  |
| 1  | HASH GROUP BY               |             | 24   | 356  |
| 2  | TABLE ACCESS BY INDEX ROWID | SALES       | 596  | 355  |
| *3 | INDEX RANGE SCAN            | SALES_DT_PR | 596  | 4    |

Nu furnizeaza randurile in ordine (2 zile diferite in interval).



Primele  $n$  inregistrari

# Rezultate parțiale

- Uneori nu sunt necesare toate liniile unei interogari SQL ci doar primele  $n$  (cateva) linii [fie pentru o afisare paginata fie pentru un “scroll infinit”].
- Asta ar putea crea probleme de performanta daca toate inregistrarile trebuie sortate pentru a fi returnate doar primele  $n$ . Din acest motiv, un “*pipelined ordered by*” este o optimizare puternica.

# Rezultate parțiale

- Un *pipelined* ORDER BY nu este atât de important din cauza că poate să returneze rândurile sortate, ci mai degrabă pentru că poate să returneze primele  $n$  rânduri fără să treacă prin toate (***pipelined* ORDER BY** are cost mic de pornire).

# Primele $n$ inregistrari

- Atunci cand se doreste preluarea primelor  $n$  inregistrari, QO nu isi da seama ca se doresc numai acele  $n$ .
- QO ar trebui sa stie daca in final aplicatia va prelua toate inregistrarile – atunci un *full scan* urmat de sortare ar fi cea mai buna solutie. Chiar si asa, un *piped order by* ar aduce un +.
- Tip: *spune-i BD cand nu ai nevoie de toate inregistrarile - fetch first.*

# Primele $n$ inregistrari

- **fetch first** – in SQL:2008, disponibil in IBM DB2, PostgreSQL, SQL Server 2012, Oracle 12c.

```
SELECT val FROM rownum_order_test
ORDER BY val DESC
FETCH FIRST 5 ROWS ONLY;
```

[Oracle 12c]

# Primele $n$ inregistrari

- Inainte (in Oracle *11g*)...
- Sa consideram urmatoarea interogare Oracle:

```
SELECT * FROM (
 SELECT *
 FROM sales
 ORDER BY sale_date DESC)
WHERE rownum <= 10;
```

# Primele $n$ inregistrari

- BD poate optimiza interogarea doar daca stie de la inceput ce rezultat partial este asteptat.
- Va prefera sa utilizeze un *pipelined order by*:

Si-a dat seama ca trebuie sa numere doar 10.

| Operation                   | Name        | Rows  | Cost |
|-----------------------------|-------------|-------|------|
| SELECT STATEMENT            |             | 10    | 9    |
| COUNT STOPKEY               |             |       |      |
| VIEW                        |             | 10    | 9    |
| TABLE ACCESS BY INDEX ROWID | SALES       | 1004K | 9    |
| INDEX FULL SCAN DESCENDING  | SALES_DT_PR | 10    | 3    |

- Nu a preluat tot ci numai 10 randuri.

# Primele $n$ inregistrari

- Utilizarea unei sintaxe corecte este numai jumătate din *job*.
- Pentru ca sa poata numara doar primele 10 inregistrari, trebuie ca neaparat sa aiba un *order by* care sa poata fi *piped* (altfel ar trebui sa sorteze tot).



# Primele $n$ inregistrari

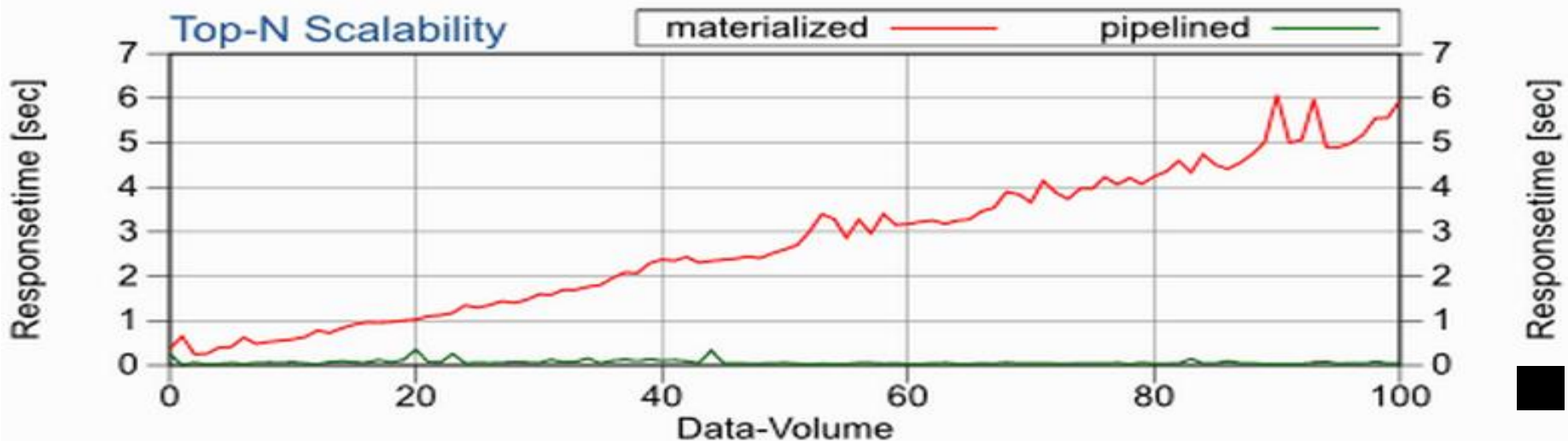
- lata ce se intampla daca nu ar exista un index care sa permita *pipeline* pentru *order by*:

| Operation             | Name  | Rows         | Cost  |
|-----------------------|-------|--------------|-------|
| SELECT STATEMENT      |       | <b>10</b>    | 59558 |
| COUNT STOPKEY         |       |              |       |
| VIEW                  |       | 1004K        | 59558 |
| SORT ORDER BY STOPKEY |       | <b>1004K</b> | 59558 |
| TABLE ACCESS FULL     | SALES | <b>1004K</b> | 9246  |

# Primele $n$ inregistrari

<http://blog.fatalmind.com/2010/09/29/finding-the-best-match-with-a-top-n-query/>

- Avantajul nu este numai in performanta imediata ci si in scalabilitatea marita. Fara *pipelined* execution, timpul creste direct proportional cu dimensiunea tabelii (cand sunt *pipelined*, timpul este **aproape** la identic).



Paginarea rezultatelor

# Paginarea rezultatelor

- Ce se intampla dupa ce s-au intors primele  $n$  rezultate ?
- Avem nevoie de rezultatele  $[n+1 .. 2n]$  (pentru urmatoarea pagina).
- Exista doua metode:
  - *offset method* (numara liniile de la inceput, filtreaza dupa row number si face discard la ce e in plus)
  - *seek method* (cauta ultima intrare din pagina precedenta si preia randurile urmatoare)

# Paginarea rezultatelor - *offset*

- Metoda offset utilizeaza un cuvânt special “**offset**” – **SQL Server** standard ca si extensie pentru `fetch first`.



```
SELECT * FROM sales
 ORDER BY sale_date DESC
 OFFSET 10 ROWS
 FETCH NEXT 10 ROWS ONLY;
```

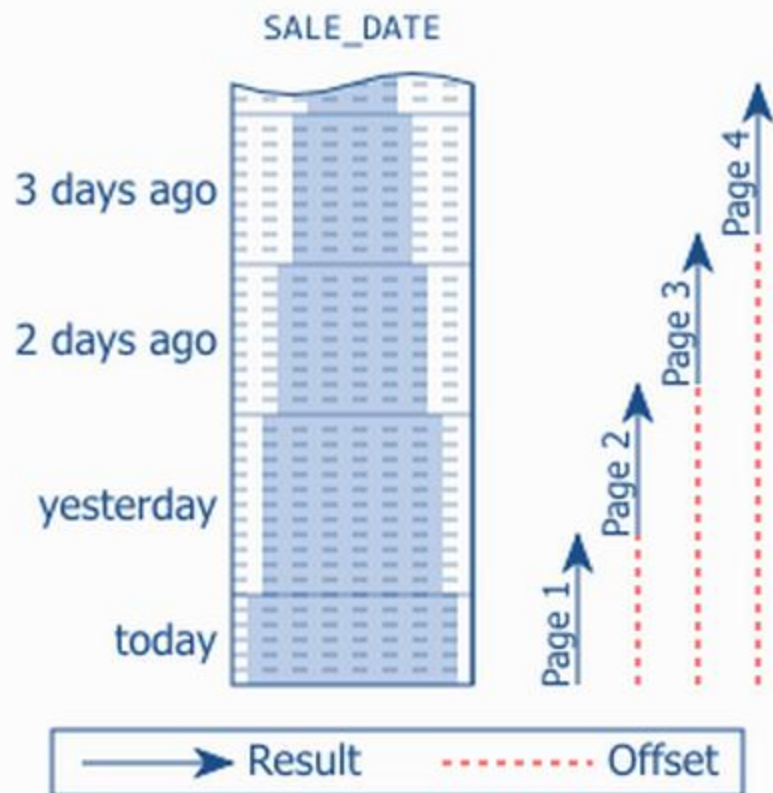
# Paginarea rezultatelor - *offset*

- In Oracle e putin mai complicat...

```
SELECT * FROM
 (SELECT tmp.*, rownum rn FROM
 (SELECT * FROM sales
 ORDER BY sale_date DESC)
 tmp WHERE rownum <= 20)
WHERE rn > 10;
```

# Paginarea rezultatelor - *offset*

- Pentru a ajunge la pagina dorita, BD trebuie sa sara peste primele pagini.



# Paginarea rezultatelor - *offset*

- Doua dezavantaje:
  - pagina se *shifteaza* cand sunt adaugate inregistrari noi (pentru ca o numara si pe cea nou adaugata);
  - timpul de raspuns este mare cand se acceseaza paginile cu numar mare.



# Paginarea rezultatelor - *seek*

- Metoda *seek* utilizeaza valorile din pagina anterioara ca punct de plecare si evita ambele dezavantaje ale metodei *offset*.
- Cauta inregistrarile care urmeaza ultimei inregistrari de pe pagina anterioara (utilizand un simplu WHERE).

# Paginarea rezultatelor - *seek*

- Presupunand ca se face doar o vanzare pe zi (deci data ar fi cheie unica):

```
SELECT * FROM sales
 WHERE sale_date < ?
 ORDER BY sale_date
DESC FETCH FIRST 10 ROWS ONLY;
```

? = Ultimul sale\_date de pe pagina precedenta

# Paginarea rezultatelor - seek

- Problema este ca daca nu avem o **cheie unica** la sfarsit de pagina, se poate intampla sa nu stim de unde incepem pagina urmatoare.
- Trebuie ca ***order by*** sa fie **determinist** altfel se poate intampla ca o noua executie sa faca un shuffle a inregistrarilor (din cauza ca QO se hotaraste sa execute altfel interogarea si atunci putem sa ne trezim ca s-a facut “*swap*” intre ultima inregistrare de pe pagina anterioara si prima de pe noua pagina).

# Paginarea rezultatelor - seek

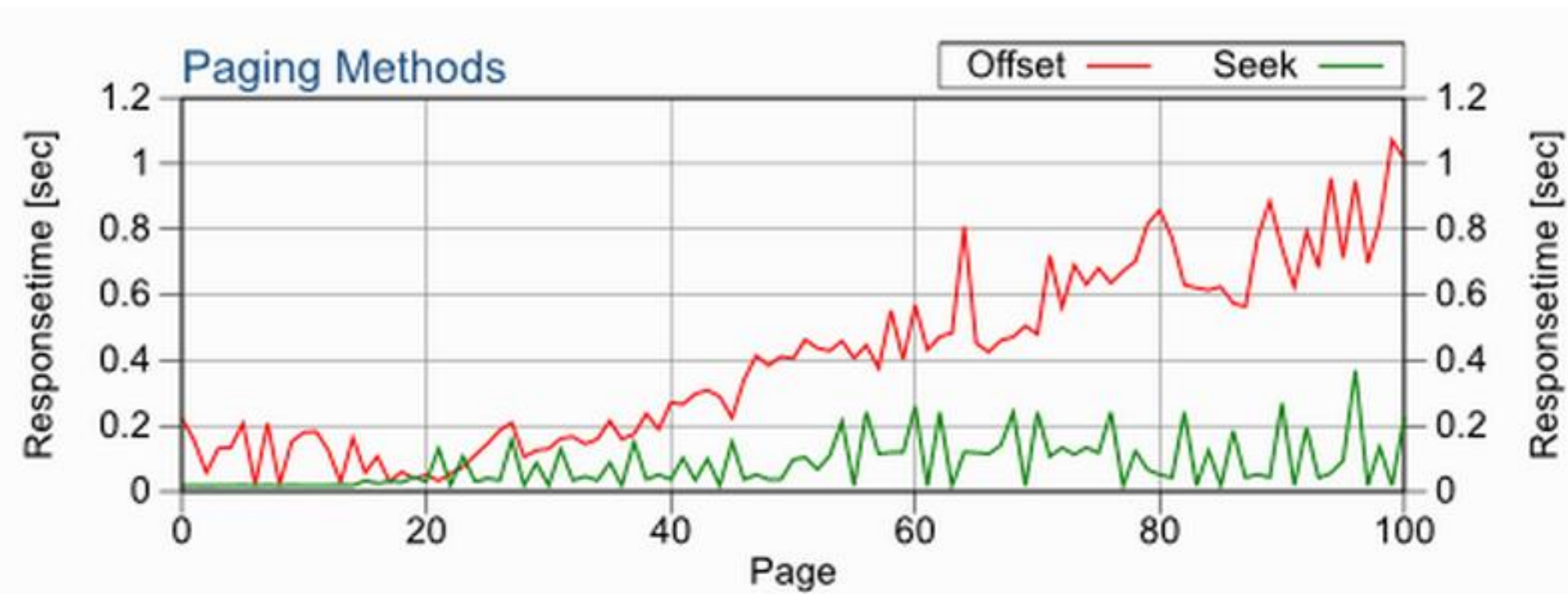
- Pentru a ne asigura ca ORDER BY este mereu determinist, am putea sa-l extindem (adaugand coloane) pana ajungem la unicate (sau daca nu se poate, se adauga un camp unic).

# Paginarea rezultatelor - seek

```
CREATE INDEX sl_dtid ON sales (sale_date,
 sale_id);
```

```
SELECT * FROM sales
 WHERE (sale_date, sale_id) < (?, ?)
 ORDER BY sale_date DESC, sale_id DESC
 FETCH FIRST 10 ROWS ONLY;
```

# Scalability offset vs seek



# Functii *Window* -> paginare

- Metoda flexibila, conform standardului.
- Numai SQL server si Oracle le utilizeaza pentru un *pipelined n-query*
- PostgreSQL nu se opreste din scanat dupa ce a preluat primele n linii 😞
- MySQL nu are deloc implementata o astfel de functie 😞

| Applies to      |
|-----------------|
| DB2<br>✗        |
| MySQL<br>✗      |
| Oracle<br>✓     |
| PostgreSQL<br>✗ |
| SQL Server<br>✓ |

# Functii *Window* -> paginare

```
SELECT * FROM (
 SELECT sales.* , ROW_NUMBER() OVER
 (ORDER BY sale_date DESC ,
 sale_id DESC) rn
 FROM sales) tmp
WHERE rn between 11 and 20
ORDER BY sale_date DESC, sale_id DESC;
```



# Functii Window -> paginare

- ROW\_NUMBER enumera randurile in functie de ordinea de sortare definita in clauza *over*.
- Ultimul *where* foloseste valoarea lui *rn* pentru a prelua doar inregistrarile 11-20.
- BD Oracle recunoaste conditia de oprire si utilizeaza indecsii SALE\_DATE si SALE\_ID pentru a produce un comportament *pipelined*

# Functii Window -> paginare

| Id | Operation                    | Name    | Rows  | Cost  |
|----|------------------------------|---------|-------|-------|
| 0  | SELECT STATEMENT             |         | 1004K | 36877 |
| *1 | VIEW                         |         | 1004K | 36877 |
| *2 | <u>WINDOW NOSORT STOPKEY</u> |         | 1004K | 36877 |
| 3  | TABLE ACCESS BY INDEX ROWID  | SALES   | 1004K | 36877 |
| 4  | INDEX FULL SCAN DESCENDING   | SL_DTID | 1004K | 2955  |

Predicate Information (identified by operation id):

- 1 - filter("RN">=11 AND "RN"<=20)
- 2 - filter(ROW\_NUMBER() OVER (  
ORDER BY "SALE\_DATE" DESC, "SALE\_ID" DESC )<=20)

# Functii Window -> paginare

Operatie de tip NOSORT = pipelined

Ce se opreste cand se ajunge la o anumita valoare (stopkey)

| Id | Operation                    |         | Rows  | Cost  |
|----|------------------------------|---------|-------|-------|
| 0  | SELECT STATEMENT             |         | 1004K | 36877 |
| *1 | VIEW                         |         | 1004K | 36877 |
| *2 | <u>WINDOW NOSORT STOPKEY</u> |         | 1004K | 36877 |
| 3  | TABLE ACCESS BY INDEX ROWID  | SALES   | 1004K | 36877 |
| 4  | INDEX FULL SCAN DESCENDING   | SL_DTID | 1004K | 2955  |

Predicate Information (identified by operation id):

- 1 - filter("RN">=11 AND "RN"<=20)
- 2 - filter(ROW\_NUMBER() OVER (  
ORDER BY "SALE\_DATE" DESC, "SALE\_ID" DESC )<=20)

# Functii Window -> paginare

- Ce le face pe functiile window sa fie importante nu este neaparat paginarea ci mai degraba calculele analitice.



Insert, Update, Delete

# Insert, Update, Delete

- Insert + Update + Delete = DML (Data Manipulation Language)
- Performanta acestor comenzi este influentata (negativ) de prezenta indecsilor.
- Indexul = date redundante. Cand sunt executate comenzile sus mentionate, indecsii trebuie refacuti si ei.

# Insert

- Insertul este influentat in primul rand de numarul indecsilor.
- Insertul nu are cum sa beneficieze de pe urma indecsilor deoarece nu are o clauza “where”.

# Insert

- Inserarea unui rand intr-o tabela presupune:
  - gasirea unei locatii pentru stocare [in tabelele fara indecsi, se cauta un block care are suficient spatiu si este adaugat acolo]
  - daca exista indecsi peste tabela, BD trebuie sa se asigure ca noua inregistrare apare in indecsi (va adauga cate o intrare in fiecare)



# Insert

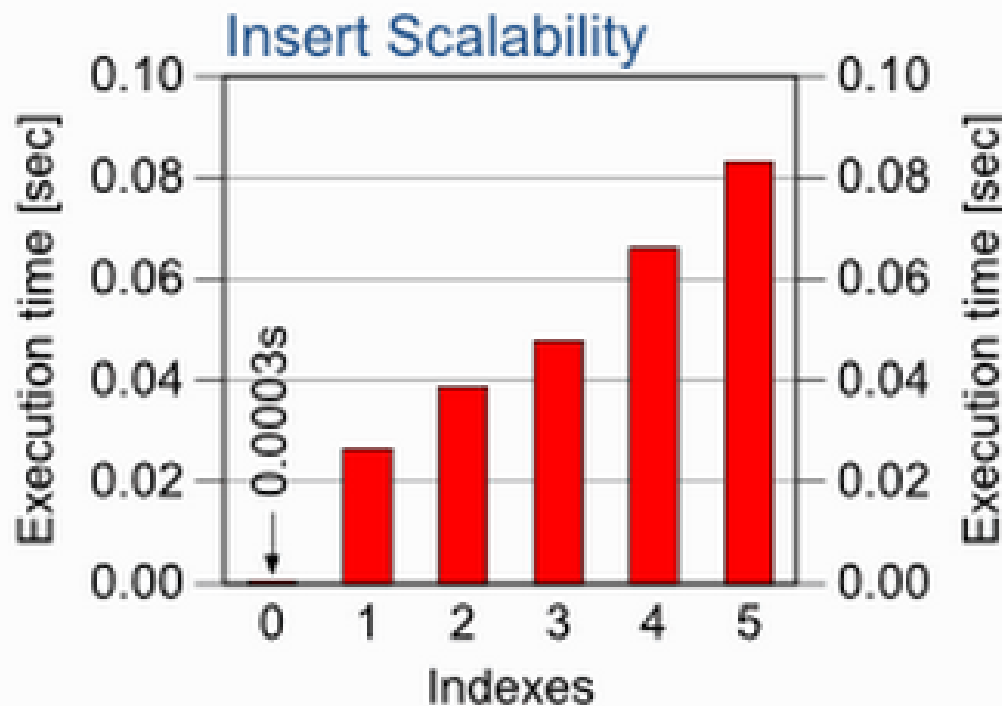
- adaugarea in index este mai costisitoare pentru ca arborele trebuie rebalansat. Din cauza existentei arborelui, inregistrarea nu poate fi scrisa in orice block; ea apartine unei anumite frunze din B-tree. Pentru a ajunge la locatie trebuie traversat arborele

# Insert

- dupa indentificarea frunzei BD confirma daca mai exista spatiu in nodul respectiv. Daca nu mai exista, nodul este impartit in doua si distribuie intrarile intre vechiul si noul nod (aici apar referinte noi si in nodul superior care la randu-i poate fi impartit). Daca toate nodurile pana la radacina sunt impartite, arborelui i se mai adauga un nivel.

# Insert

- Timpul unui insert in functie de numarul de indecsi:



# Insert

- Primul index face cea mai mare diferenta
- Considerand ca intr-o tabela s-ar face numai inserturi, e mai bine fara indecsi.
- **Performanta fara indecsi este atat de buna ca uneori are sens sa renuntam la indecsi atunci cand incarcam cantitati mari de date (indecsii nu trebuie refacuti in timpul incarcarii ci numai o singura data la final).**

# Insert

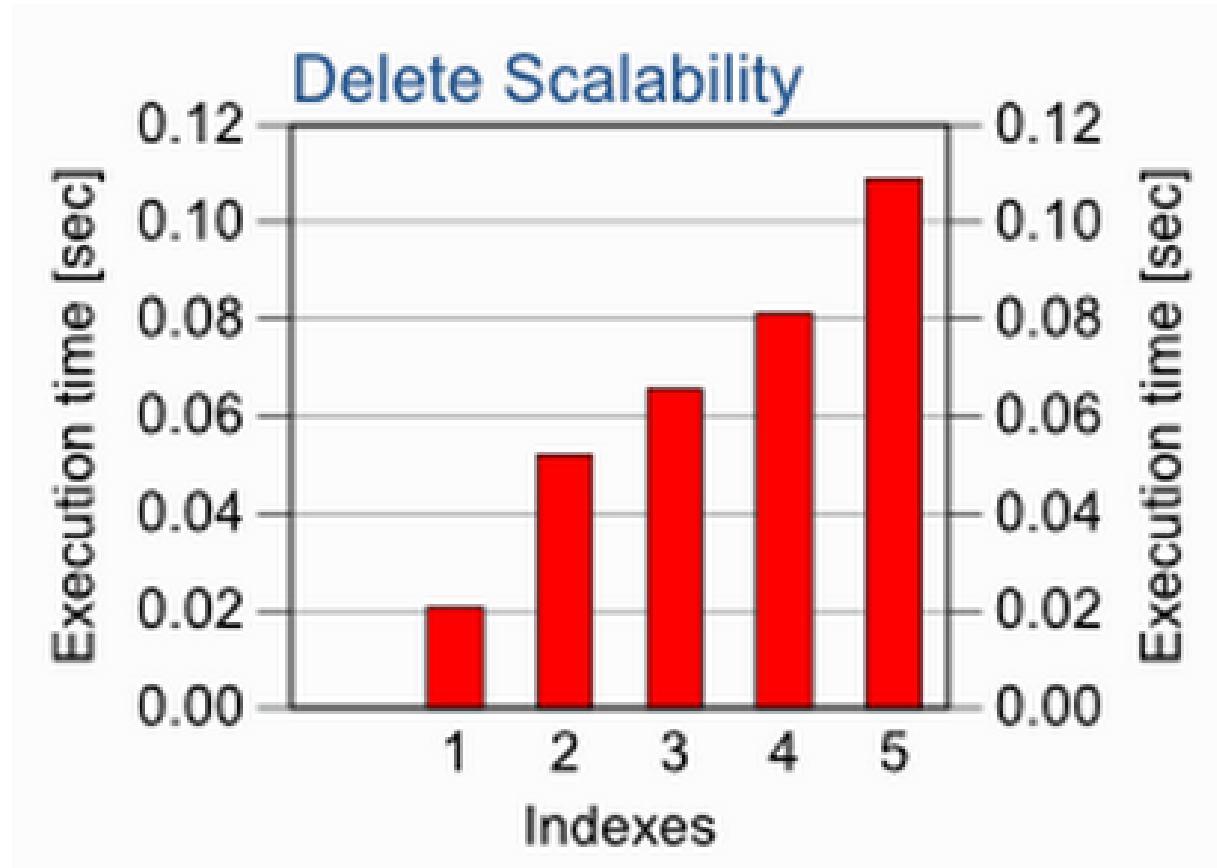
- Cum s-ar proceda in cadrul unei “index organized table” sau “clustered index” ?

# Delete

- Delete are o clauza “where” care o face sa beneficieze de pe urma indecsilor.  
Funtioneaza ca un select urmat de stergerea datelor.
- Stergerea unui rand este inversa fata de inserarea unui rand.

# Delete

- Eficienta operatiei delete este reprezentata in tabelul:



# Delete

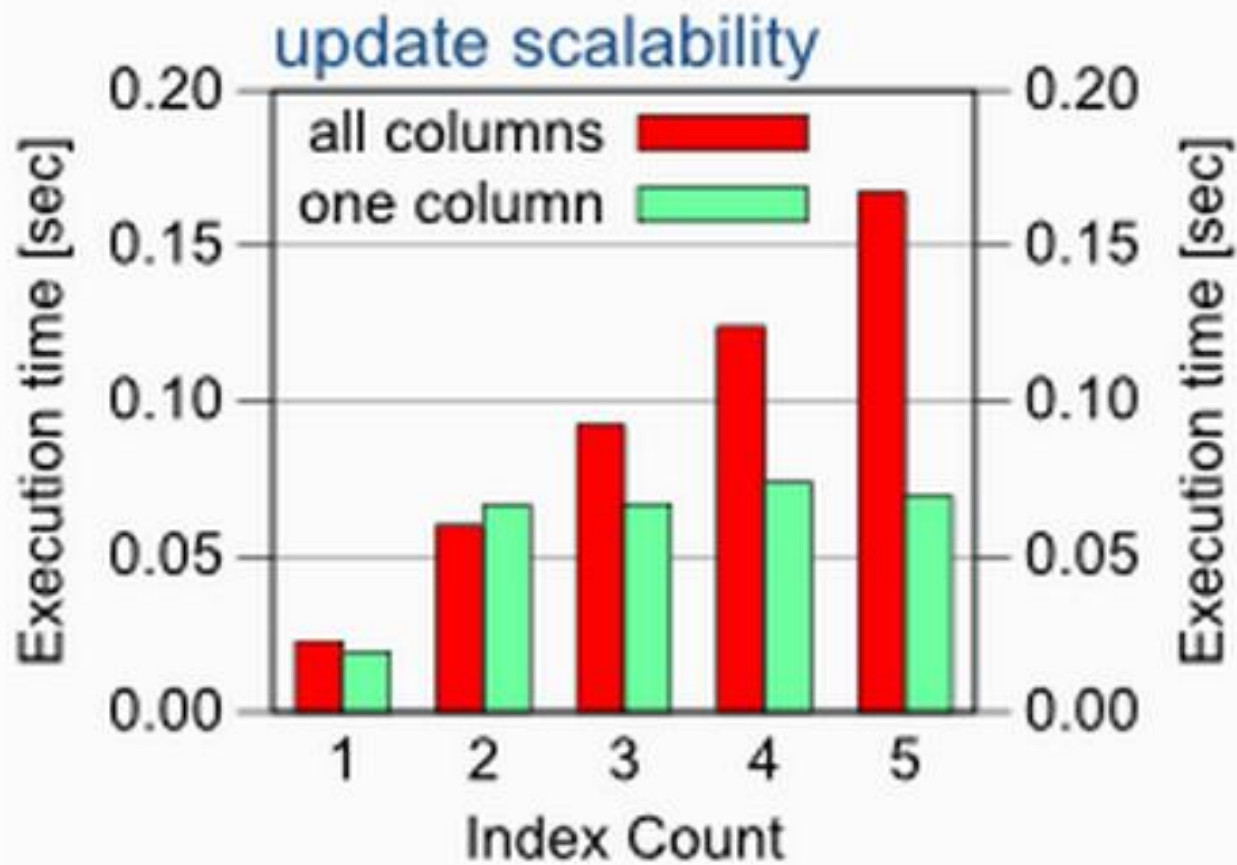
- Daca nu exista index, BD trebuie sa faca *full table scan* pentru a cauta inregistrarea.
- Are sens sa stergem fara “where” atunci cand sunt sterse foarte multe randuri.
- Operatiile delete si update au un execution plan.



# Update

- O operatie de tip Update trebuie sa realoce indexul (sa stearga vechea pozitie si sa realoce informatia in alta pozitie).
- Update (la fel ca insert si delete) depinde de numarul indecsilor peste tabela.
- Daca updateul nu modifica campurile din indecsi, timpul este minim (pentru ca indecsii nu trebuie refacuti).

# Update



# Update

- Daca este afectat un singur index, timpul ramane acelasi chiar daca sunt mai multi indecsi peste tabela.
- Cand se scrie comanda update, trebuie avut grija sa fie modificate doar acele coloane care trebuie modificate si nu toate coloanele (utilizand valori vechi in update).

Ganditi-va la un caz cand insert sau delete nu afecteaza toti indecsii dintr-o tabela. Raspuns:



# Bibliografie (online)

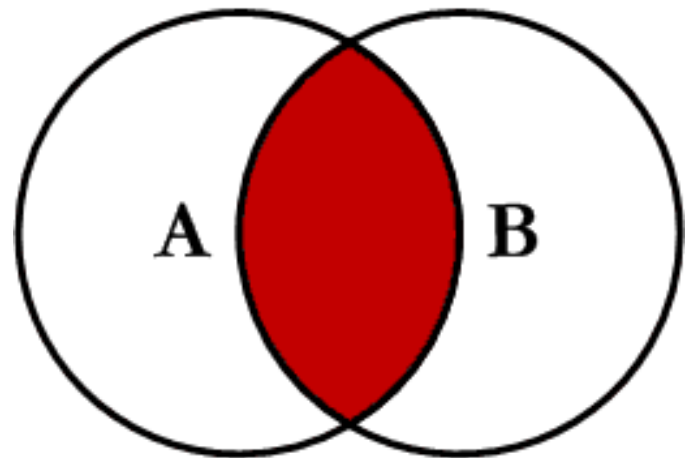
- <http://use-the-index-luke.com/>

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

# Joins – INNER JOIN

- <http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins>
- Cel mai simplu tip de join, interogarea va returna toate inregistrarile din A care au inregistrari pereche in tabela B.

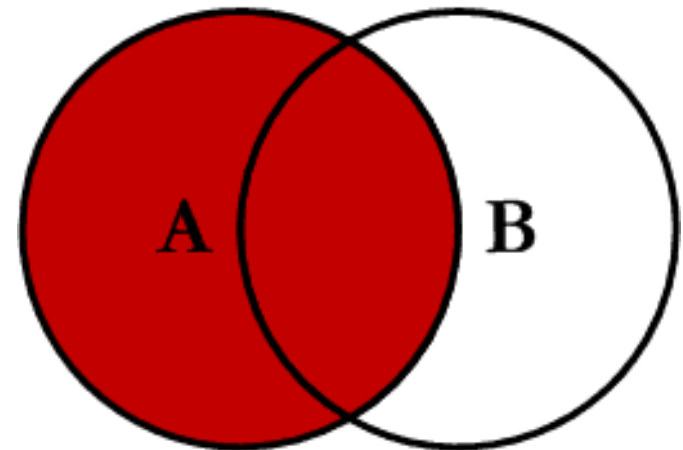
```
SELECT <select_list>
FROM Table_A A
INNER JOIN Table_B B
ON A.Key = B.Key
```



# Joins – LEFT JOIN

- Returnează înregistrările din tabela A indiferent dacă au un match în tabela B. Dacă în tabela B sunt înregistrări care fac match, le va potrivi.

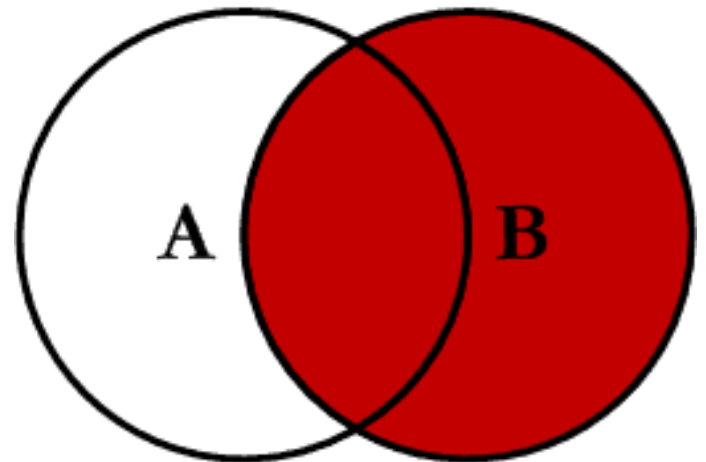
```
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON A.Key = B.Key
```



# Joins – RIGHT JOIN

- Returneaza inregistrarile din tabela B indiferent daca au un match in tabela A. Daca in tabela A sunt inregistrari care fac match, le va potrivi.

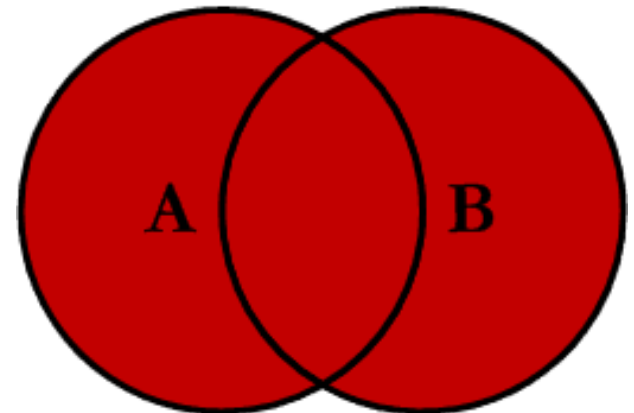
```
SELECT <select_list>
FROM Table_A A
RIGHT JOIN Table_B B
ON A.Key = B.Key
```



# Joins – OUTER JOIN

- Va intoarce inregistrarile care se potrivesc in ambele tabele (A si B).

```
SELECT <select_list>
 FROM Table_A A
 FULL OUTER JOIN Table_B B
 ON A.Key = B.Key
```

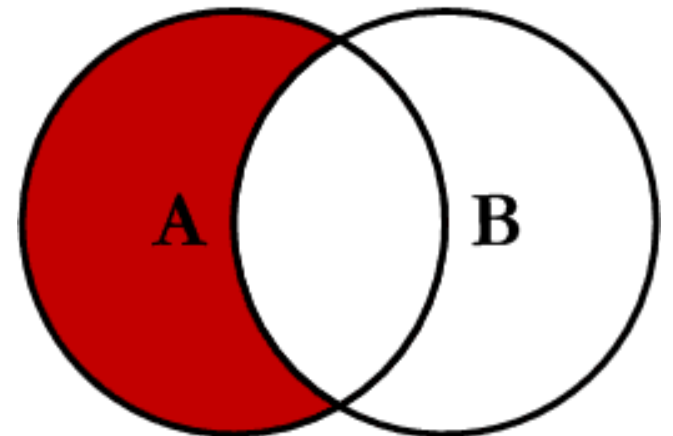




# Joins – LEFT EXCLUDING JOIN

- Va intoarce inregistrarile din A care nu au un match in B.

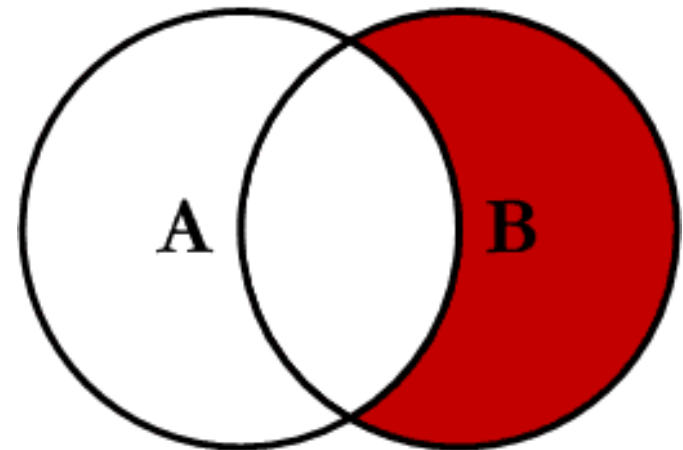
```
SELECT <select_list>
 FROM Table_A A
 LEFT JOIN Table_B B ON A.Key = B.Key
WHERE B.Key IS NULL
```



# Joins – RIGHT EXCLUDING JOIN

- Va intoarce inregistrarile din B care nu au un match in A.

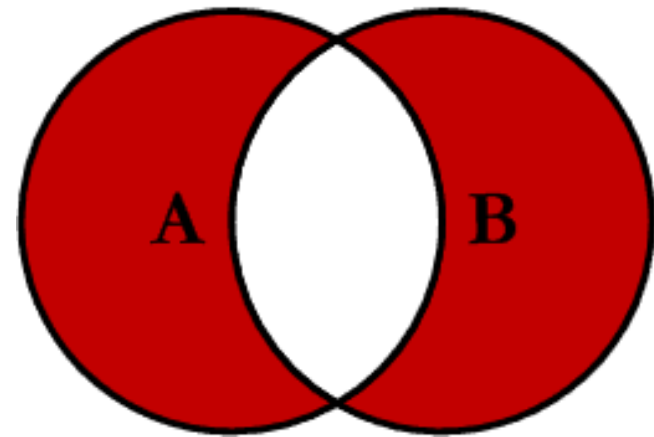
```
SELECT <select_list>
 FROM Table_A A
 RIGHT JOIN Table_B B
 ON A.Key = B.Key
 WHERE B.Key IS NULL
```



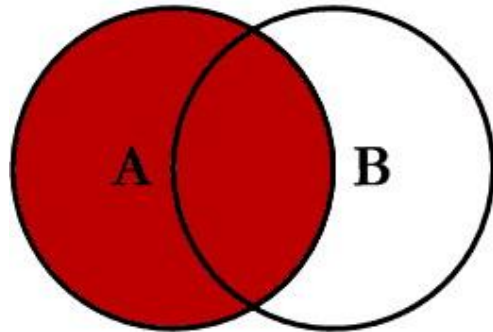
# Joins – OUTER EXCLUDING JOIN

- Va intoarce inregistrarile din A si din B care nu se potrivesc.

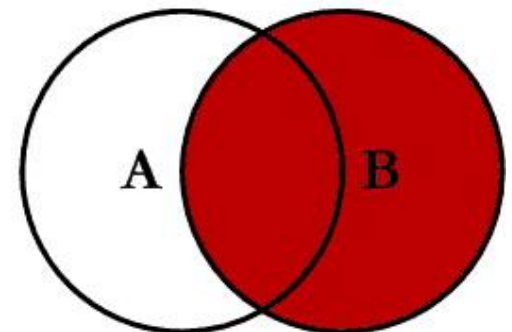
```
SELECT <select_list>
 FROM Table_A A FULL OUTER JOIN Table_B B
 ON A.Key = B.Key
 WHERE A.Key IS NULL OR B.Key IS NULL
```



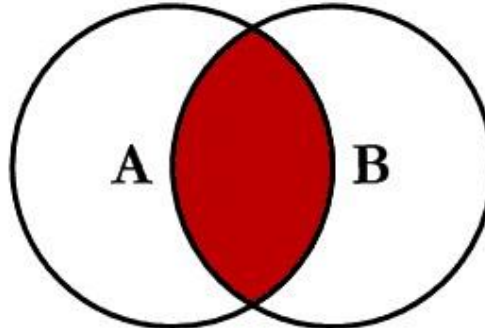
# SQL JOINS



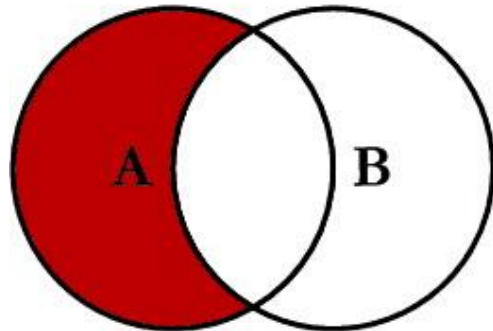
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



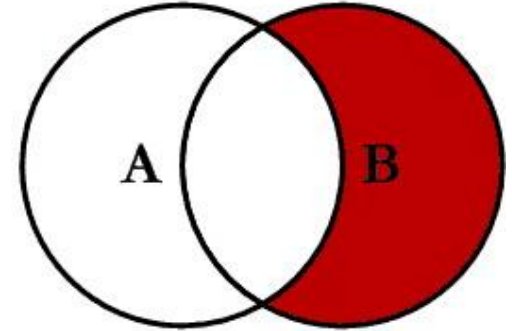
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



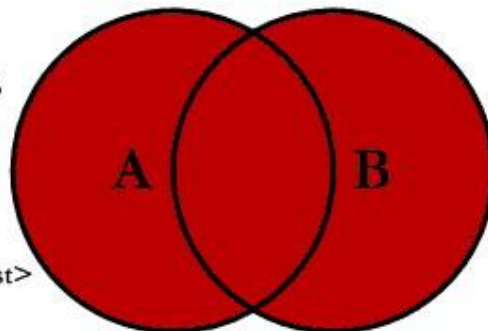
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



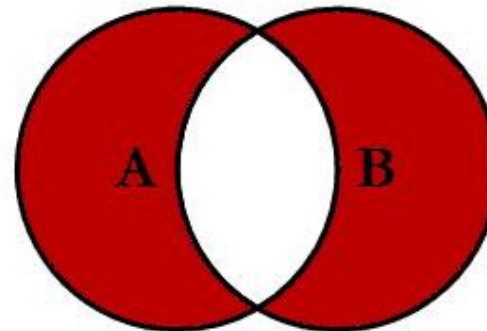
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
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
```