Bases de données avancées
Chapitre 4 : *Database Tuning*

Avec les slides de

© Dennis Shasha © Philippe Bonnet

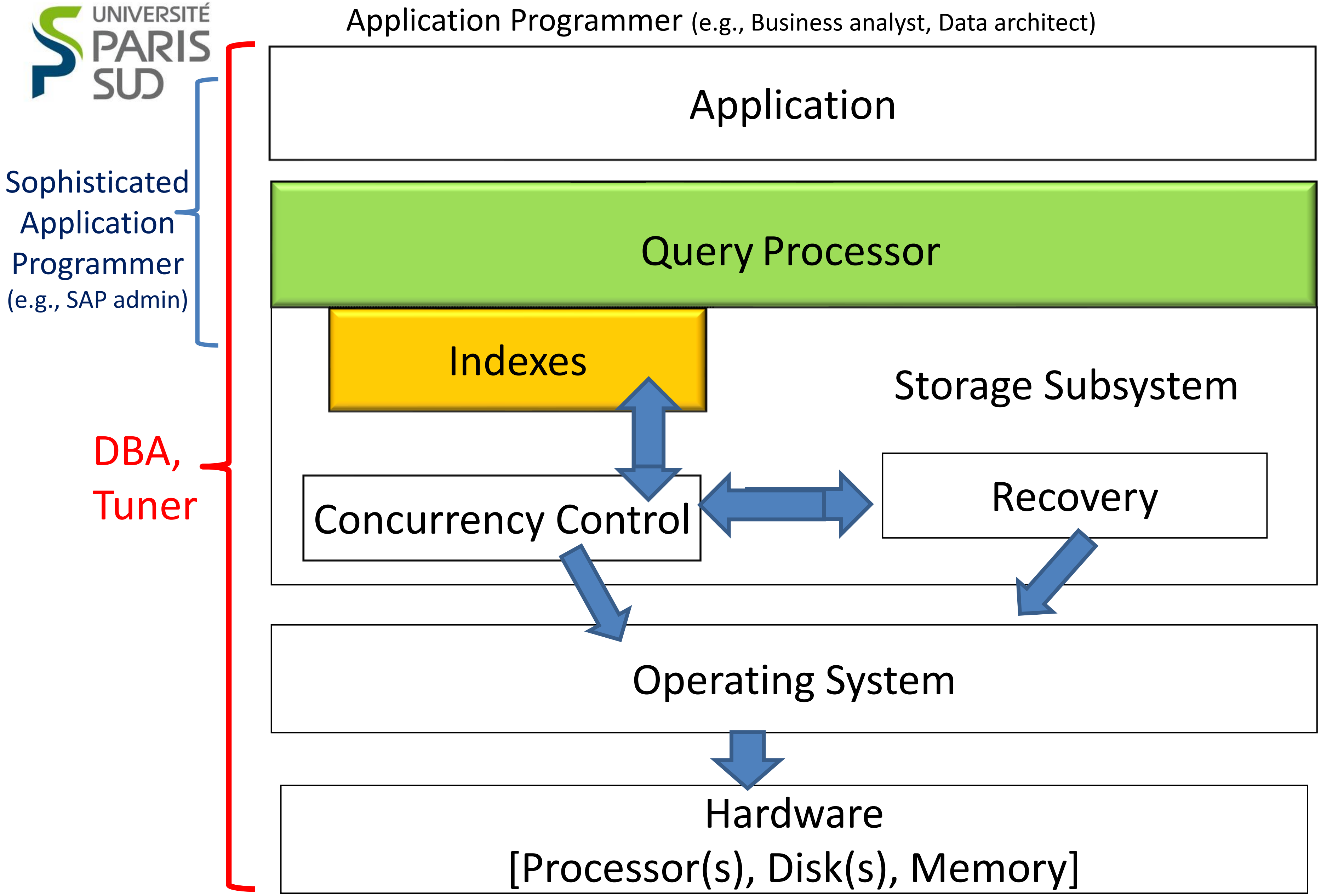
Sarah Cohen-Boulakia
Laboratoire de Recherche en Informatique
Université Paris Sud
http://www.lri.fr/~cohen/BD/BD.html

Database Tuning

Database Tuning is the activity of making a database application run more quickly. "More quickly" usually means higher throughput, though it may mean lower response time for time-critical applications.

Tuning Principles *Leitmotifs*

- Think globally, fix locally (does it matter?)
- Partitioning breaks bottlenecks (temporal and spatial)
- Start-up costs are high; running costs are low (disk transfer, cursors)
- Be prepared for trade-offs (indexes and inserts)



Application Programmer (e.g., Business analyst, Data architect)
Application
Query Processor
Indexes
Storage Subsystem
Concurrency Control
Recovery
Operating System
Hardware
[Processor(s), Disk(s), Memory]
Sophisticated Application Programmer (e.g., SAP admin)
DBA, Tuner

Part 1 : Index Tuning

- Index issues
  - Indexes may be better or worse than scans
  - Multi-table joins that run on for hours, because the **wrong indexes are defined**
  - Concurrency control bottlenecks
  - Indexes that are **maintained and never used**

Index Implementations in some major DBMS (change quickly!)

- SQL Server
  - **B+-Tree** data structure
  - Clustered indexes are sparse
  - Indexes maintained as updates/insertions/deletes are performed
- DB2
  - **B+-Tree** data structure, spatial extender for R-tree
  - Clustered indexes are dense
  - Explicit command for index reorganization
- Oracle
  - **B+-tree**, hash, bitmap, spatial extender for R-Tree
  - clustered index
    - Index organized table (unique/clustered)
    - Clusters used when creating tables.
- TimesTen (Main-memory DBMS)
  - **T-tree**

B+-Tree Performance

- Key length is important!
  - Choose **small key** when creating an index
  - Key compression techniques in DBMS
    - Prefix compression (Oracle 8, mySQL): only store that part of the key that is needed to distinguish it from its neighbors: Smi, Smo, Smy for Smith, Smoot, Smythe.
    - Front compression (Oracle 5): adjacent keys have their front portion factored out: Smi, (2)o, (2)y. There are problems with this approach:
      - Processor overhead for maintenance
      - Locking Smoot requires locking Smith too.

Types of Queries

- **Point Query**

  SELECT balance
  FROM accounts
  WHERE **number = 1023**;

// Number is a primary key

- **Multipoint Query**

  SELECT balance
  FROM accounts
  WHERE **branchnum = 100**;

// several matches

- **Range Query**

  SELECT number
  FROM accounts
  WHERE **balance > 10000 and balance <= 20000**;

- **Prefix Match Query**

  SELECT *
  FROM employees
  WHERE **name = 'J*'** ;

More Types of Queries

- **Extremal Query**

  SELECT *
  FROM accounts
  WHERE balance =
  (select **max**(balance) from accounts)

- **Ordering Query**

  SELECT *
  FROM accounts
  ORDER BY balance;

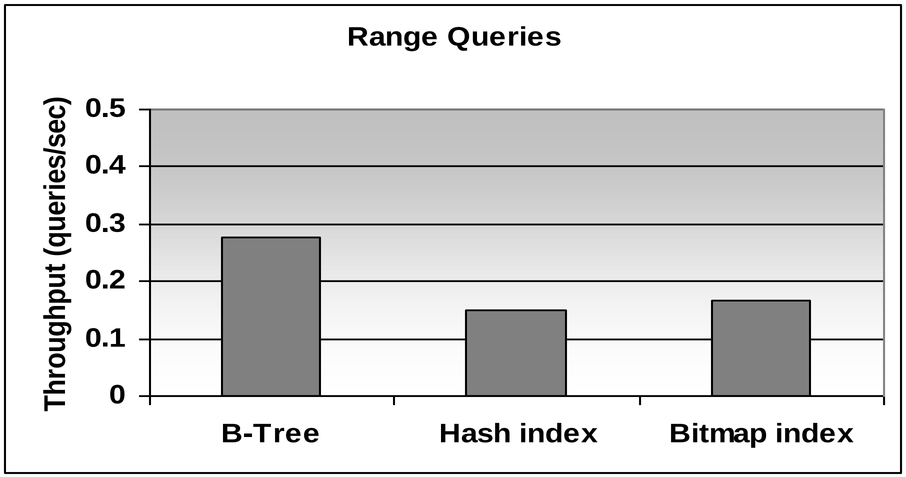
- **Grouping Query**

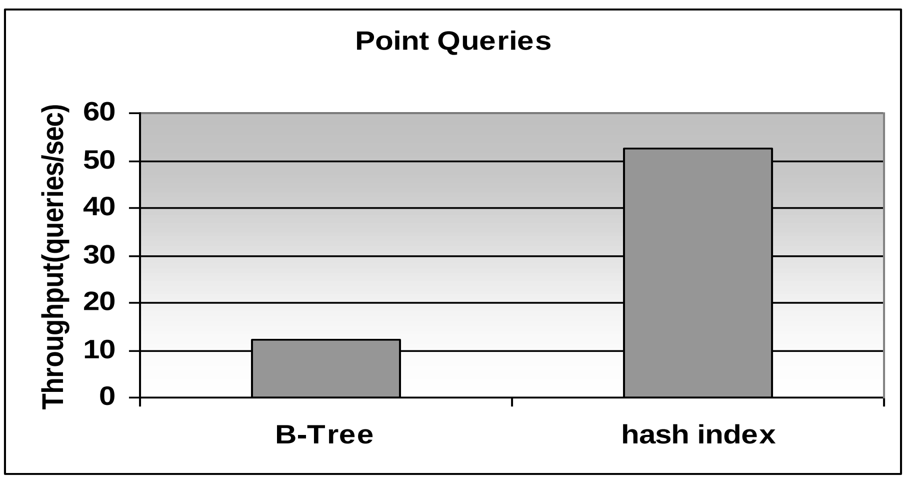
  SELECT branchnum, avg(balance)
  FROM accounts
  GROUP BY branchnum;

- **Join Query**

  SELECT distinct branch.adresse
  FROM accounts, branch
  WHERE
  accounts.branchnum = branch.number
  and accounts.balance > 10000;

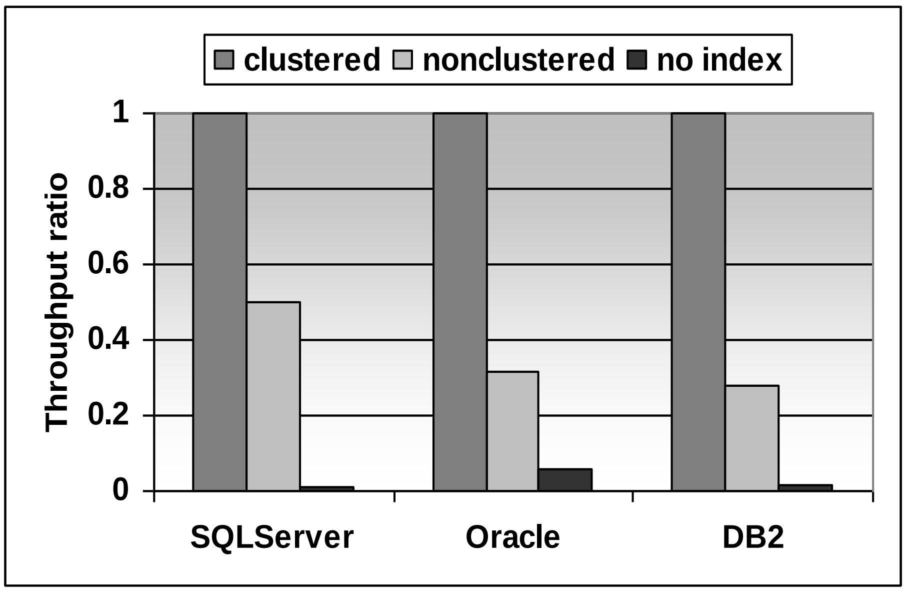
B-Tree, Hash Tree, Bitmap





- Hash indexes don't help when evaluating range queries
- Hash index outperforms B-tree on point queries

Clustered Index

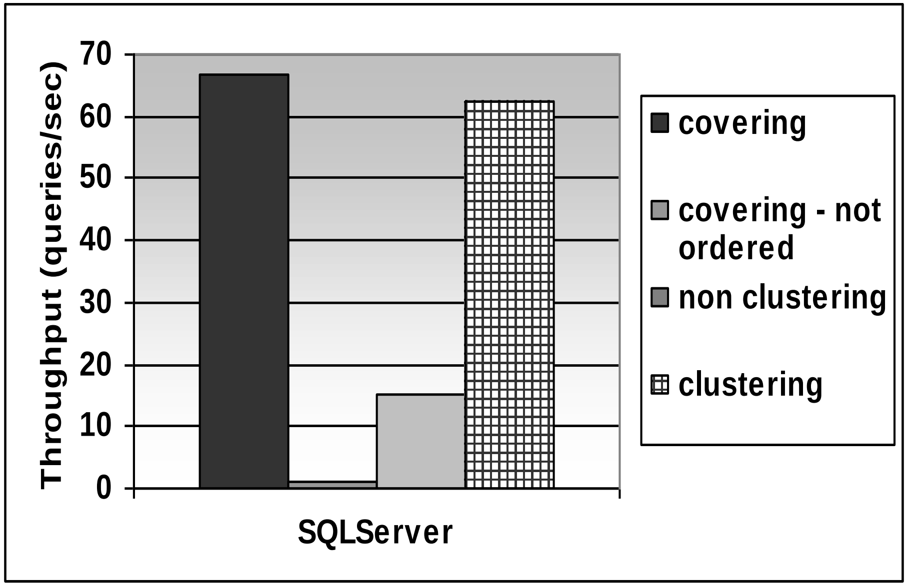


- **Multipoint query** that returns 100 records out of 1000 000 (0,01%).
- Clustered index is twice as fast as non-clustered index and orders of magnitude faster than a scan.

Covering Index - defined

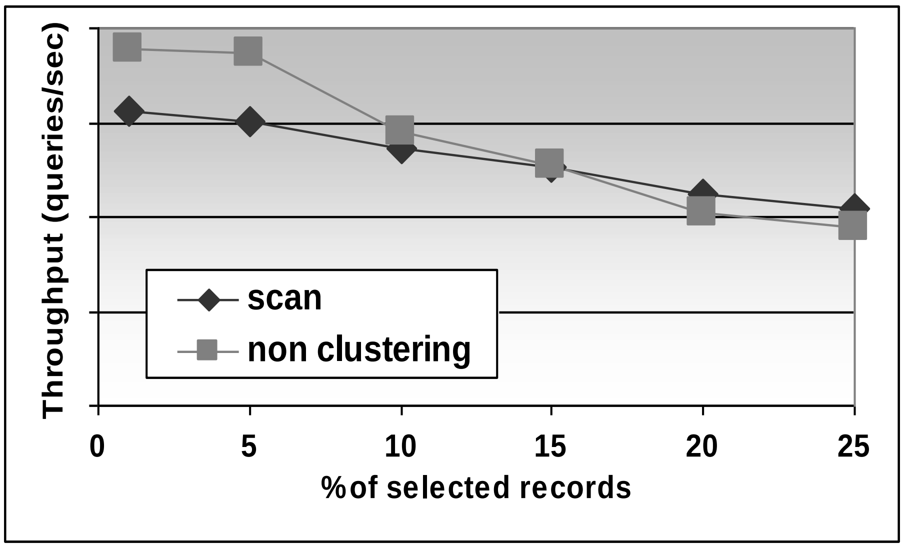
- Select name from employee where department = “marketing”
- Good covering index would be on (department, name)
- Index on (name, department) not useful.
- Index on department alone moderately useful.

Covering Index - impact



- Covering index performs better than clustering index when first attributes of index are in the where clause and last attributes in the select.
- When attributes are not in order then performance is much worse.

Scan Can Sometimes Win

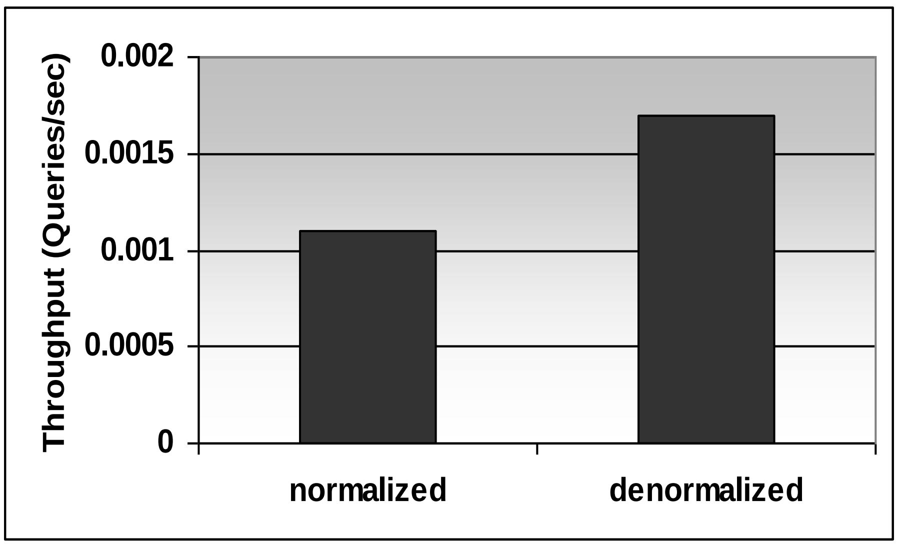


- IBM DB2 v7.1 on Windows 2000
- Range Query
- If a query retrieves 10% of the records or more, scanning is often better than using a non-clustering non-covering index. Crossover > 10% when records are large or table is fragmented on disk – scan cost increases.

Part 2 : Schema tuning

- Normalisation & Denormalisation
- Vertical partitioning

Denormalization



- Query: find all lineitems whose supplier is in Europe.
- With a normalized schema this query is a 4-way join.
- If we denormalize lineitem and add the name of the region for each lineitem (foreign key denormalization) throughput improves 30%

Vertical Partitioning

- Consider account(id, balance, homeaddress)
- When might it be a good idea to do a “vertical partitioning” into account1(id,balance) and account2(id,homeaddress)?
- Join vs. size.

Vertical Partitioning

- Which design is better depends on the query pattern:
  - The application that sends a monthly statement is the principal user of the address of the owner of an account
  - The balance is updated or examined several times a day.
- The second schema might be better because the relation (account_ID, balance) can be made smaller:
  - More account_ID, balance pairs fit in memory, thus increasing the hit ratio
  - A scan performs better because there are fewer pages.

Tuning Normalization

- A single normalized relation XYZ is better than two normalized relations XY and XZ if the single relation design allows queries to access X, Y and Z together without requiring a join.
- The two-relation design is better iff:
  - Users access tend to partition between the two sets Y and Z most of the time
  - Attributes Y or Z have large values

Part 3 : Query Tuning

- **Query optimisation**

→EXPLAIN ANALYSE

- **Query rewriting**

Query Tuning

```
SELECT  s.RESTAURANT_NAME, t.TABLE_SEATING, to_char(t.DATE_TIME,'Dy, Mon FMDD') AS THEDATE, to_char(t.DATE_TIME,'HH:MI PM')
AS THETIME,to_char(t.DISCOUNT,'99') || '%' AS AMOUNTVALUE,t.TABLE_ID, s.SUPPLIER_ID, t.DATE_TIME,
to_number(to_char(t.DATE_TIME,'SSSSS')) AS SORTTIME

FROM TABLES_AVAILABLE t, SUPPLIER_INFO s,

        (SELECT     s.SUPPLIER_ID, t.TABLE_SEATING, t.DATE_TIME, max(t.DISCOUNT) AMOUNT, t.OFFER_TYPE
         FROM      TABLES_AVAILABLE t, SUPPLIER_INFO
         WHERE     t.SUPPLIER_ID = s.SUPPLIER_ID
           and (TO_CHAR(t.DATE_TIME, 'MM/DD/YYYY') !=
                  TO_CHAR(sysdate, 'MM/DD/YYYY') OR TO_NUMBER(TO_CHAR(sysdate, 'SSSSS')) < s.NOTIFICATION_TIME - s.TZ_OFFSET)
           and t.NUM_OFFERS  > 0
           and t.DATE_TIME > SYSDATE
           and s.CITY = 'SF'
           and t.TABLE_SEATING = '2'
           and t.DATE_TIME between sysdate and (sysdate + 7)
           and to_number(to_char(t.DATE_TIME, 'SSSSS')) between 39600 and 82800
           and t.OFFER_TYPE = 'Discount'
        GROUP BY
         s.SUPPLIER_ID, t.TABLE_SEATING, t.DATE_TIME, t.OFFER_TYP
       ) u
WHERE
         t.SUPPLIER_ID   = s.SUPPLIER_ID
        and u.SUPPLIER_ID   = s.SUPPLIER_ID
        and t.SUPPLIER_ID   = u.SUPPLIER_ID
        and t.TABLE_SEATING = u.TABLE_SEATING
        and t.DATE_TIME     = u.DATE_TIME
        and t.DISCOUNT = u.AMOUNT
        and t.OFFER_TYPE    = u.OFFER_TYPE
        and (TO_CHAR(t.DATE_TIME, 'MM/DD/YYYY') !=
                  TO_CHAR(sysdate, 'MM/DD/YYYY') OR
                  TO_NUMBER(TO_CHAR(sysdate, 'SSSSS'))  < s.NOTIFICATION_TIME - s.TZ_OFFSET)
        and t.NUM_OFFERS    >
        and t.DATE_TIME > SYSDATE and s.CITY = 'SF' and t.TABLE_SEATING = '2'  and t.DATE_TIME between sysdate and (sysdate + 7) and
                  to_number(to_char(t.DATE_TIME, 'SSSSS')) between 39600 and 82800 and t.OFFER_TYPE = 'Discount'

ORDER  BY AMOUNTVALUE DESC,  t.TABLE_SEATING ASC, upper(s.RESTAURANT_NAME) ASC,SORTTIME ASC, t.DATE_TIME ASC
```

Execution is too slow …

1) How is this query executed?
2) How to make it run faster?

→ EXPLAIN

Query Execution Plan

Output of the Oracle EXPLAIN tool

Execution Plan

```
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=165 Card=1 Bytes=106)
   1    0   SORT (ORDER BY) (Cost=165 Card=1 Bytes=106)
   2    1     NESTED LOOPS (Cost=164 Card=1 Bytes=106)
   3    2       NESTED LOOPS (Cost=155 Card=1 Bytes=83)
   4    3         TABLE ACCESS (FULL) OF 'TABLES_AVAILABLE' (Cost=72 Card=1 Bytes=28)
   5    3         VIEW
   6    5           SORT (GROUP BY) (Cost=83 Card=1 Bytes=34)
   7    6             NESTED LOOPS (Cost=81 Card=1 Bytes=34)
   8    7               TABLE ACCESS (FULL) OF 'TABLES_AVAILABLE' (Cost=72 Card=1 Bytes=24)
   9    7               TABLE ACCESS (FULL) OF 'SUPPLIER_INFO' (Cost=9  Card=20 Bytes=200)
  10    2       TABLE ACCESS (FULL) OF 'SUPPLIER_INFO' (Cost=9 Card=20 Bytes=460)
```

Physical Operators

Access Method

Cost Model

Physical Operators

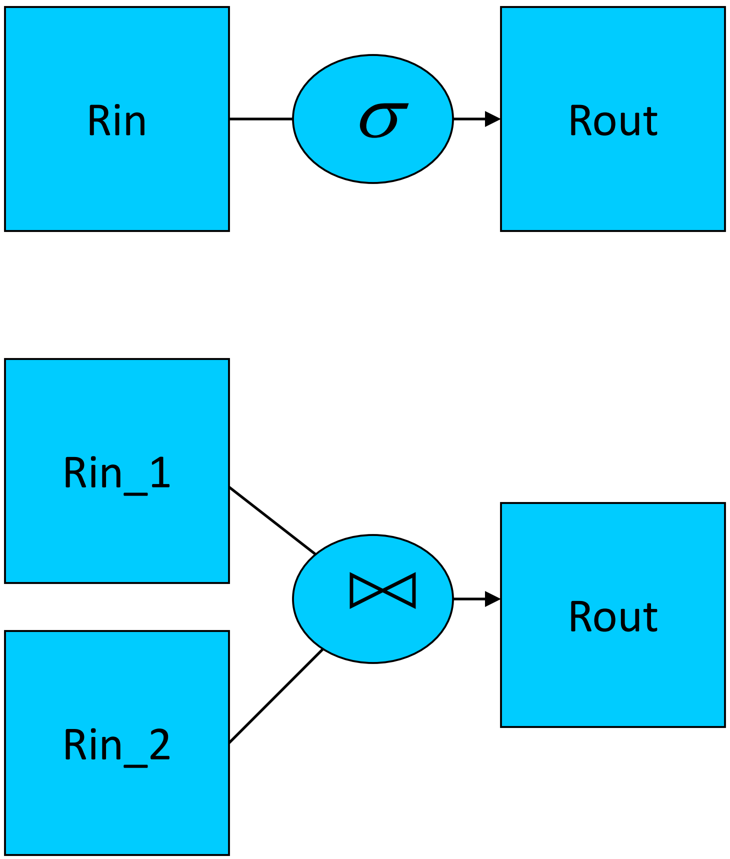
- Query Blocks
  - One block per SELECT-FROM-WHERE-GROUPBY-ORDERBY
  - VIEW isolate blocks optimized separately
- Shape of the execution tree (right-deep, bushy, …)
- Join order

- Algorithms
  - Sort
  - Aggregates
  - Select
  - Project
  - Join
    - Nested Loop
    - Sort-Merge
    - Hash-Join

Access Method

- Table Scan (*full scan*)
- Index Scan
  - Find Index(es) matching expression in query
  - Extract constant or range from query
  - Index Search

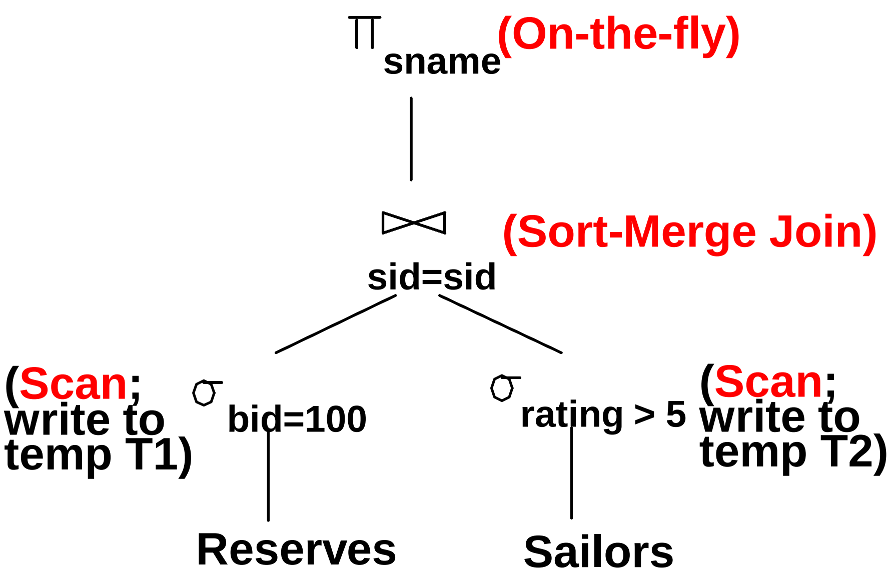
Cost Model

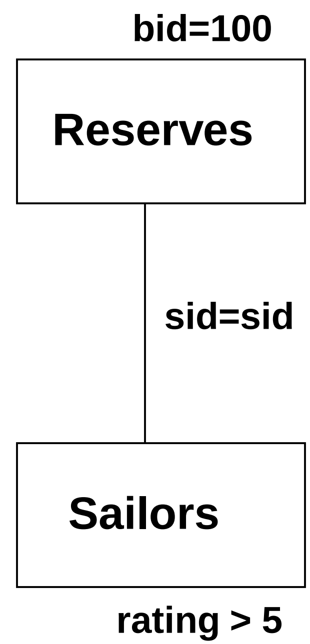


- Cost metric
  - Cost = w1*IO_COST +w2*CPU_COST
  - We consider w2 = 0
- Cost formula for each operator
  - Depends on operator algorithm
  - Depends on input size (nb tuples, nb pages)
    - Because operators are composed. Need to estimate size of operator output.

Query Representation

- Query Tree
- Query graph





Query Representation

- A query is decomposed into blocks
  - Aggregation
  - Order by
  - SPJ
  - Relations
- Each block is represented and optimized independently

Overview of Query Optimization

- Ideally: Want to find best plan.
- Practically: Avoid worst plans!
- Two main issues:
  - For a given query, what is the search space?
  - How is the search implemented?
    - Algorithm to search plan space for cheapest (estimated) plan.
    - How is the cost of a plan estimated?

Search Algorithm

Naïve1

- Enumerate all possible plans (o(n!))
- Pick the best plan
- Intractable

Naïve 2

- Order of relations fixed by the query
- Selections are pushed
  - No further transformations
- Single multiway nested loop join for each block
  - Index used if they exist
  - Star tree

Search Algorithm

Semi-Naïve

- Order of relations fixed by the query
- Selections are pushed
  - No further transformations
- Nested loop vs. sort merge join
  - Left-deep tree

Implementation problems:

- expressions reference columns of tables
- expressions must be adapted to the position of tables in the tree (including interm. tables)

Part 3 : Query Tuning

- **Query optimisation**

→EXPLAIN, ANALYSE

- **Query rewriting**

Query Rewriting

The first tuning method to try is the one whose effects are purely local

- Adding an index, changing the schema, modifying transactions have global effects that are potentially harmful
- Query rewriting only impacts a particular query

Query Rewriting Techniques

- Index usage
- DISTINCTs elimination
- (Correlated) subqueries
- Use of temporaries (no query in the FROM clause!)
- Join conditions
- Use of Having
- Use of views
- Materialized views.

Running Example

- Employee(<u>ssnum</u>, name, manager, dept, salary, numfriends)
  - Clustering index on ssnum
  - Non clustering indexes (i) on name and (ii) on dept
  - Ssnum determines all the other attributes
- Student(<u>ssnum</u>, name, degree_sought, year)
  - Clustering index on ssnum
  - Non clustering index on name
  - Ssnum determines all the other attributes
- Tech(<u>dept</u>, manager, location)
  - Clustering index on dept; dept is primary key.

Index Usage

- Many query optimizers **will not use indexes** in the presence of:
  - **Arithmetic expressions**

    WHERE salary/12 >= 4000;
  - **Substring expressions**

    SELECT * FROM employee
      WHERE SUBSTR(name, 1, 1) = ‘G’;
  - Numerical comparisons of fields with **different types**
  - **Comparison with NULL**.

Eliminate unneeded DISTINCTs

- Query: Find employees who work in the information systems department. There should be no duplicates.

```
SELECT distinct ssnum
FROM employee
WHERE dept = ‘information systems’
```

- Distinct needed ?

Eliminate unneeded DISTINCTs

- Query: Find social security numbers of employees in the technical departments. There should be no duplicates.

```
SELECT DISTINCT ssnum
FROM employee, tech
WHERE employee.dept = tech.dept
```

- Is DISTINCT needed?

Reaching

- The relationship among DISTINCT, keys and joins can be generalized:
  - Call a table T *privileged* if the fields returned by the SELECT contain a key of T
  - Let R be an unprivileged table. Suppose that R is joined on equality by its key field to some other table S, then we say R *reaches* S.
  - Now, define reaches to be transitive. So, if R1 reaches R2 and R2 reaches R3 then say that R1 reaches R3.

Reaches: Main Theorem

- There will be no duplicates among the records returned by a selection, even in the absence of DISTINCT if one of the two following conditions hold:
  - Every table mentioned in the FROM clause is privileged
  - Every unprivileged table reaches at least one privileged table.

Reaches: Example 1

```
SELECT ?DISTINCT? ssnum
FROM employee, tech
WHERE employee.manager = tech.manager
```

Reaches: Example 2

```
SELECT ?DISTINCT? ssnum, tech.dept
FROM employee, tech
WHERE employee.manager = tech.manager
```

Reaches: Example 3

```
SELECT student.ssnum
FROM student, employee, tech
WHERE student.name = employee.name
    AND employee.dept = tech.dept;
```

Rewriting of Uncorrelated Subqueries without Aggregates

1. Combine the arguments of the two FROM clauses
2. AND together the where clauses, replacing in by =
3. Retain the SELECT clause from the outer block

```
SELECT ssnum
FROM employee
WHERE dept in (select dept
from tech)
```

becomes

```
SELECT ssnum
FROM employee, tech
WHERE employee.dept =
                    tech.dept
```

NB: one dept per employee (possible iff “in” meant “=“)

Abuse of Temporaries

- Query: Find all information department employees with their locations who earn at least $10,000.
  - INSERT INTO temp
    SELECT *
    FROM employee
    WHERE salary >= 10000
  - SELECT ssnum, location
    FROM temp
    WHERE temp.dept = ‘information systems’

  Or same idea with temp in the FROM clause
- Selections should have been done in reverse order. Temporary relation blinded the optimizer.

Join Conditions

- It is a good idea to express join conditions on clustering indexes.
  - No sorting for sort-merge.
  - Speed up for multipoint access using an indexed nested loop.
- It is a good idea to express join conditions on <u>numerical attributes rather than on string attributes.</u>

Use of Having

- Don't use HAVING when WHERE is enough!
  - SELECT avg(salary) as avgsalary,
    dept
    FROM employee
    GROUP BY dept
    HAVING dept = 'information systems';
  - SELECT avg(salary) as avgsalary,
    dept
    FROM employee
    WHERE dept= 'information systems'
    GROUP BY dept;
- Having should be reserved for **aggregate properties of the groups**.
  - SELECT avg(salary) as avgsalary, dept
    FROM employee
    GROUP BY dept
    HAVING count(ssnum) > 100;

Tuning Queries and Views (Conclusion)

- If a query runs slower than expected, check if an index needs to be re-built or if statistics are too old (→ ANALYSE).
- Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:
  - Selections involving null values
  - Selections involving arithmetic or string expressions
  - Selections involving OR conditions
  - Lack of evaluation features like index-only strategies or certain join methods or poor size estimation
- Check the plan that is being used! Then adjust the choice of indexes or rewrite the query/view

→ EXPLAIN

→ EXPLAIN ANALYSE