# Query tuning

## Viet-Trung Tran
## SoICT

# Outline

- Course organization
- Introduction to database tuning
- Basic principles of tuning

# What is query tuning

- Rewrite query to run faster
- First thing to do if query is slow
- Other tuning approaches related to query
  - Adding indexes
  - Changing schema ( 3, 4 NF, etc)
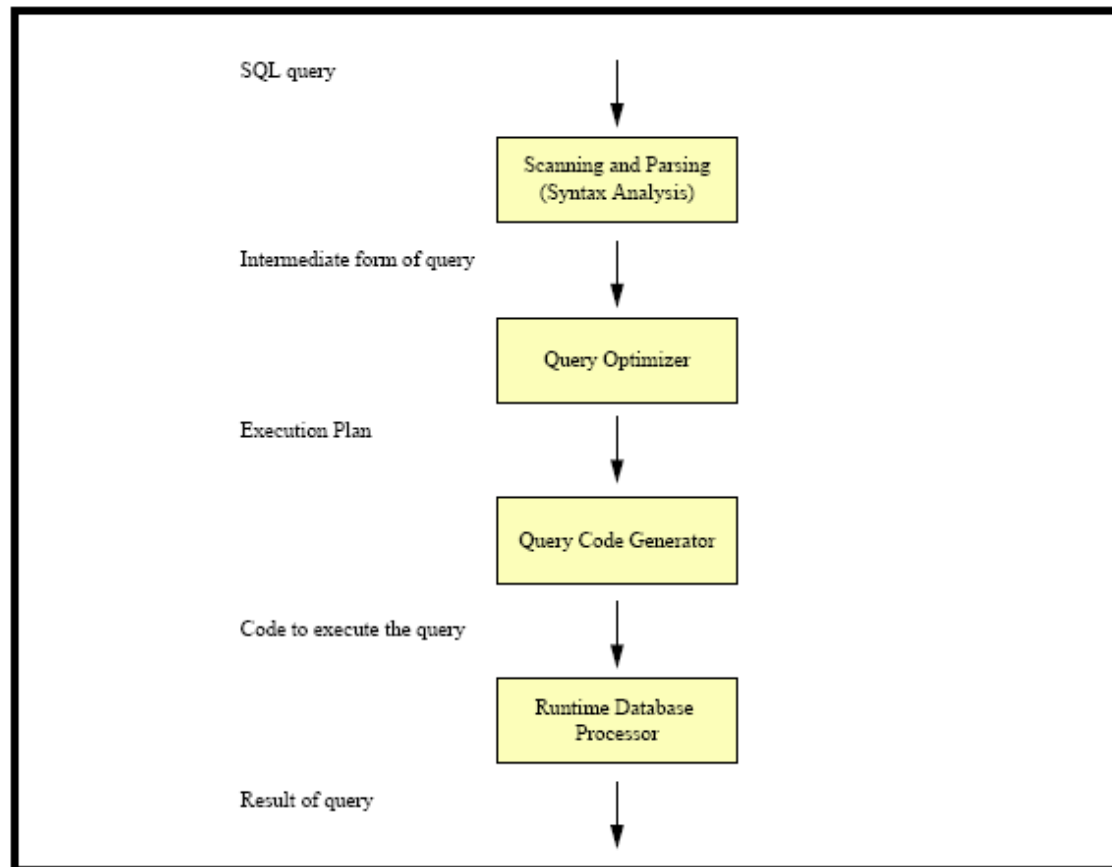  - Modify transaction lengh

# Query processing: review



Figure 2

# Paser

- Input: SQL query
  - SELECT balance FROM account
    WHERE balance < 2500
- Output: Relational algebra expression

$$\sigma_{balance<2500}(\Pi_{balance}(account))$$

- But it's not unique

$$\Pi_{balance}(\sigma_{balance<2500}(account))$$

# Optimizer

- Input: Relational algebra expression

$$\Pi_{balance}\left(\sigma_{balance<2500}\left(account\right)\right)$$

- Output: Query plan
  - Selection (optimization) process
    - Equivalence transformation
    - Annotation of the relation algebra expression
    - Cost estimation for different query plans

$\Pi_{balance}$

|

$\sigma_{balance<2500}$
*use index 1*

|

*account*
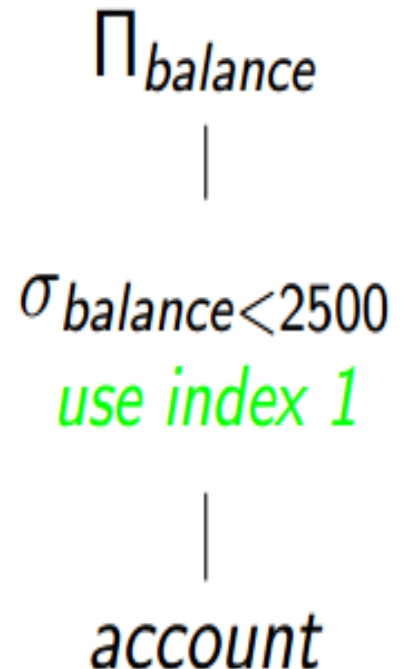
# Step 1:Equivalence Transformation

- Equivalence of relational algebra expressions:
  - equivalent if they generate the same set of tuples on every legal
    database instance
  - legal: database satisfies all integrity constraints specified in the database schema
- Equivalence rules:
  - transform one relational algebra expression into equivalent one
  - similar to numeric algebra: $a + b = b + a$, $a(b + c) = ab + ac$, etc.
- Why producing equivalent expressions?
  - equivalent algebraic expressions give the same result
  - but usually the execution time varies significantly

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections; *cascade of* $\sigma$.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the final operations in a sequence of projection operations is needed, the others can be omitted; *cascade of* $\Pi$

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins:

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$

$$\sigma_{\theta_1}(E_1 \bowtie_{\sigma_{\theta_2}} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

5. Theta join operations are commutative:

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. Natural-join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

Theta joins are associative in the following manner

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where $\theta_2$ involves attributes from $E_2$ and $E_3$ only.

7. The selection operation distributes over the theta join operation under the following two conditions:

(a) It distributes when all the attributes in the selection condition $\theta_0$ involve only the attributes of one of the expressions ($E_1$) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

(b) It distributes when the selection condition $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta join.

(a) Let $L_1$ and $L_2$ be attributes of $E_1$ and $E_2$ respectively. Suppose that the join condition $\theta$ involves only attributes in $L_1 \cup L_2$. Then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

(b) Consider a join $E_1 \bowtie_\theta E_2$. Let $L_1$ and $L_2$ be sets of attributes from $E_1$ and $E_2$ respectively. Let $L_3$ be attributes of $E_1$ that are involved in the join condition $\theta$, but are not in $L_1 \cup L_2$, and let $L_4$ be attributes of $E_2$ that are involved in the join condition $\theta$, but are not in $L_1 \cup L_2$. Then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set-difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2 = \sigma_P(E_1) - \sigma_P(E_2)$$

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

① Conjunctive selections can be deconstructed into a sequence of individual selections:

$$\sigma_{p_1 \wedge p_2}(E) \quad \equiv \quad \sigma_{p_1}(\sigma_{p_2}(E))$$

② Selection operations are commutative:

$$\sigma_{p_1}(\sigma_{p_2}(E)) \quad \equiv \quad \sigma_{p_2}(\sigma_{p_1}(E))$$

③ Only the last projection in a sequence of projections is needed, the others can be omitted:

$$\pi_{L_1}(\pi_{L_2}(\cdots \pi_{L_n}(E) \cdots)) \quad \equiv \quad \pi_{L_1}(E)$$

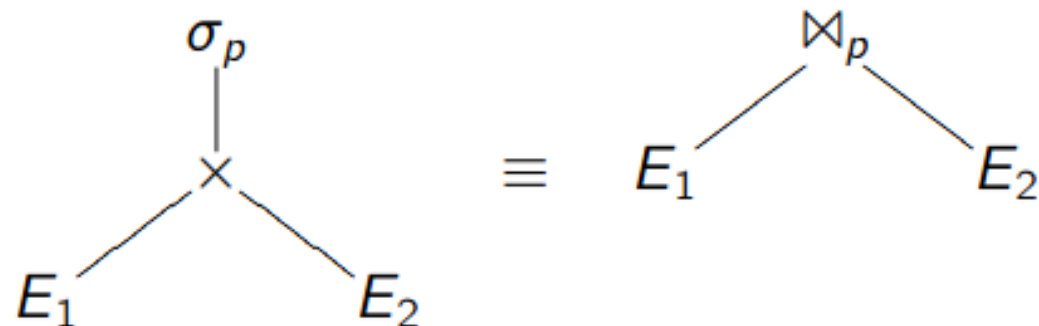④ Selections can be combined with cartesian prodcuts and joins:

(a)
$$\sigma_p(E_1 \times E_2) \quad \equiv \quad E_1 \bowtie_p E_2$$

(b)
$$\sigma_p(E_1 \bowtie_q E_2) \quad \equiv \quad E_1 \bowtie_{p \wedge q} E_2$$

▶ Pictorial description of ④ (a):

⑤ Join operations are commutative:

$$E_1 \bowtie_p E_2 \quad \equiv \quad E_2 \bowtie_p E_1$$

⑥ (a) Natural joins (equality of common attributes) are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 \quad \equiv \quad E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Generals joins are associative in the following sense:

$$(E_1 \bowtie_p E_2) \bowtie_{q \wedge r} E_3) \quad \equiv \quad E_1 \bowtie_{p \wedge q} (E_2 \bowtie_r E_3)$$

where predicate $r$ involves attributes of $E_2$, $E_3$ only.

⑦ Selection distributes over joins in the following ways:

(a) If predicate $p$ involves attributes of $E_1$ only:

$$\sigma_p(E_1 \bowtie_q E_2) \quad \equiv \quad \sigma_p(E_1) \bowtie_q E_2$$

(b) If predicate $p$ involves only attributes of $E_1$ and $q$ involves only attributes of $E_2$:

$$\sigma_{p \wedge q}(E_1 \bowtie_r E_2) \quad \equiv \quad \sigma_p(E_1) \bowtie_r \sigma_q(E_2)$$

(this is a consequence of rules ⑦ (a) and ①).

⑧ Projection distributes over join as follows:

$$\pi_{L_1 \cup L_2}(E_1 \bowtie_p E_2) \equiv \pi_{L_1}(E_1) \bowtie_p \pi_{L_2}(E_2)$$

if $p$ involves attributes in $L_1 \cup L_2$ only and $L_i$ contains attributes of $E_i$ only.

⑨ The set operations union and intersection are commutative:

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

⑩ The set operations union and intersection are associative:

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

⑪ The selection operation distributes over $\cup, \cap$ and $\setminus$:

$$
\begin{aligned}
\sigma_p(E_1 \cup E_2) &\equiv \sigma_p(E_1) \cup \sigma_p(E_2) \\
\sigma_p(E_1 \cap E_2) &\equiv \sigma_p(E_1) \cap \sigma_p(E_2) \\
\sigma_p(E_1 \setminus E_2) &\equiv \sigma_p(E_1) \setminus \sigma_p(E_2)
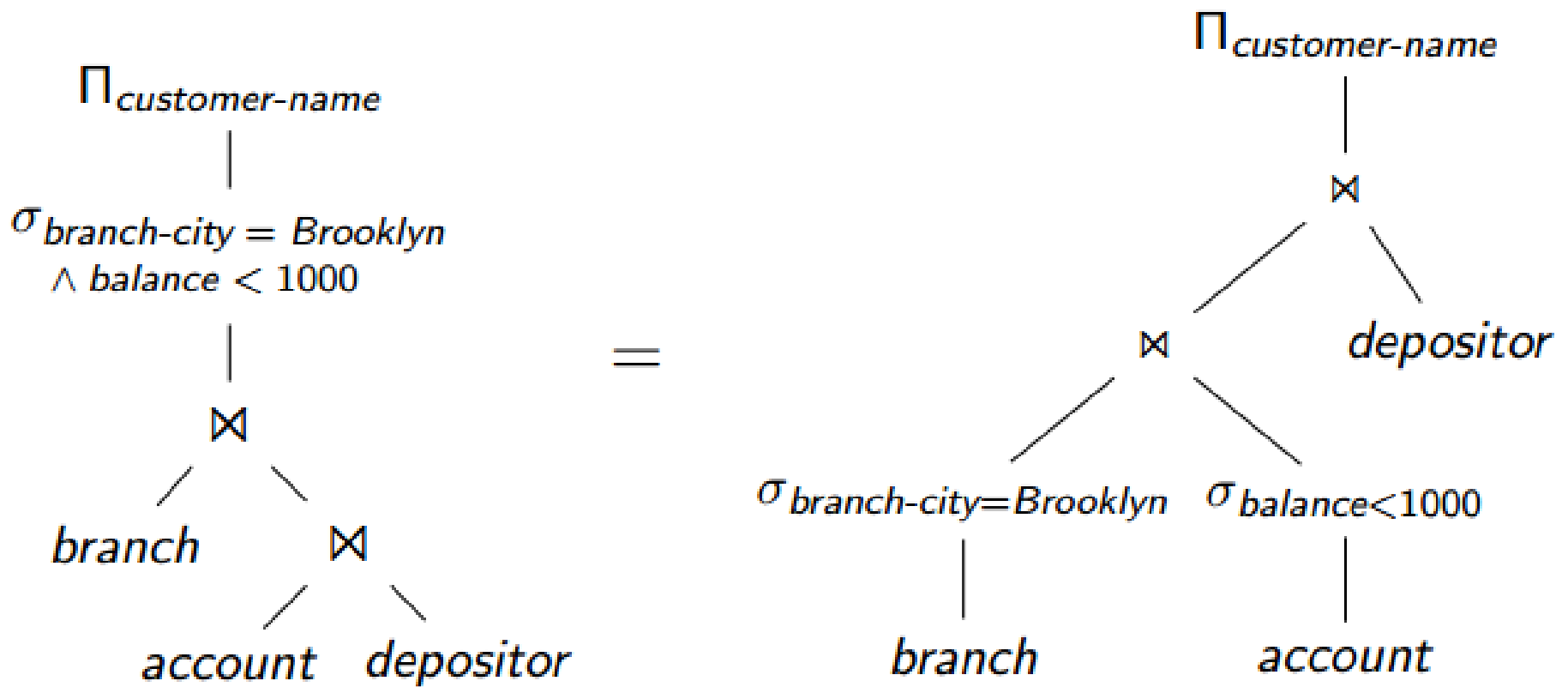\end{aligned}
$$

Also:

$$
\begin{aligned}
\sigma_p(E_1 \cap E_2) &\equiv \sigma_p(E_1) \cap E_2 \\
\sigma_p(E_1 \setminus E_2) &\equiv \sigma_p(E_1) \setminus E_2
\end{aligned}
$$

(this does not apply for $\cup$ )

⑫ The projection operation distributes over $\cup$:

$$
\pi_L(E_1 \cup E_2) \equiv \pi_L(E_1) \cup \pi_L(E_2)
$$

$$\Pi_{customer\text{-}name}$$

$$|$$

$$\sigma_{branch\text{-}city\,=\,Brooklyn} \\ \wedge\ balance\ <\ 1000$$

$$|$$

$$\bowtie$$

branch    $\bowtie$

account   depositor

$$=$$

$$\Pi_{customer\text{-}name}$$

$$|$$

$$\bowtie$$

$$\bowtie$$    depositor

$$\sigma_{branch\text{-}city=Brooklyn}\qquad \sigma_{balance<1000}$$

branch                    account

# Heuristic rules

▶ Query optimizers use the equivalence rules of relational algebra to improve the expected performance of a given query in *most cases*.

▶ The optimization is guided by the following **heuristics**:

(a) **Break apart conjunctive selections** into a sequence of simpler selections

(rule ①—preparatory step for (b)).

(b) **Move $\sigma$ down the query tree** for the earliest possible execution

(rules ②, ⑦, ⑪—reduce number of tuples processed).

(c) **Replace $\sigma-\times$ pairs by $\bowtie$**

(rule ④ (a)—avoid large intermediate results).

(d) **Break apart and move as far down the tree as possible lists of projection attributes**, create new projections where possible

(rules ③, ⑧, ⑫—reduce tuple widths early).

(e) **Perform the joins with the smallest expected result first.**
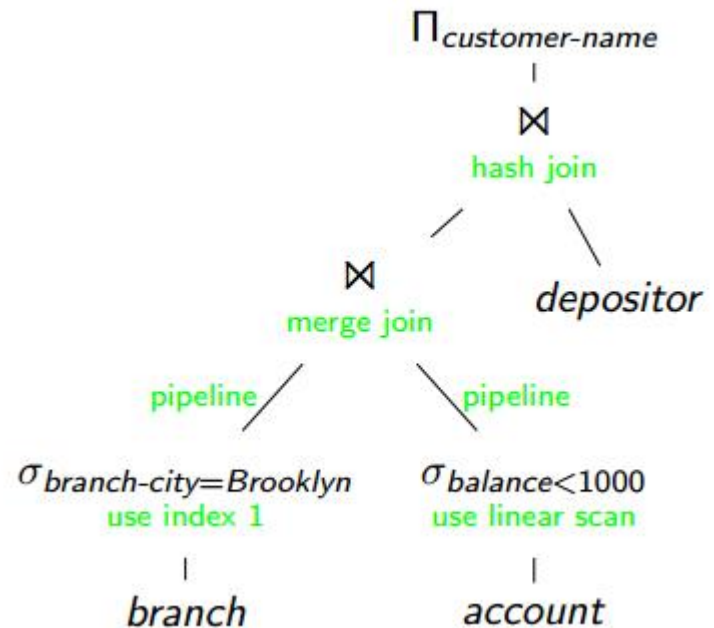
# References

- http://www.cs.iusb.edu/~hhakimza/B561/Lecture/Part_3f_Query_Processing_Optimization.pdf

# Step 2 : Annotation – Making query plan

- Algebra expression is not a query plan.
- Additional decisions required:
    - which indexes to use, for example, for joins and selects?
    - which algorithms to use, for example, sort-merge vs. hash join?
    - materialize intermediate results or pipeline them? etc.
- Each relational algebra expression can result in many query plans.
- Some query plans may be better than others!

# Example

- query plan of our example query:
  - account physically sorted by branch-name;
  - index 1 on branch-city



$\Pi_{customer-name}$
|
$\bowtie$
hash join

$\bowtie$
merge join

$depositor$

pipeline

pipeline

$\sigma_{branch-city=Brooklyn}$
use index 1
|
$branch$

$\sigma_{balance<1000}$
use linear scan
|
$account$

# Step 3: Cost estimation

- Finding the fastest one
  - Just an estimation under certain assumptions
  - Huge number of query plans may exist

# Cost estimation factors

- Catalog information: database maintains statistics about relations
- Ex.
  - number of tuples per relation
  - number of blocks on disk per relation
  - number of distinct values per attribute
  - histogram of values per attribute
- Problems
  - cost can only be estimated
  - updating statistics is expensive, thus they are often out of date

# Choosing the cheapest query plan

- Problem: Estimating cost for all possible plans too expensive.
- Solutions:
  - pruning: stop early to evaluate a plan
  - heuristics: do not evaluate all plans
- Real databases use a combination:
  - Apply heuristics to choose promising query plans.
  - Choose cheapest plan among the promising plans using pruning.
- Examples of heuristics:
  - perform selections as early as possible
  - perform projections early avoid Cartesian products

# Execution engine

- receives query plan from optimizer
- executes plan and returns query result to user

# Why query tuning? Why query optimizer is not enough?

- Optimizers are not perfect:
  - transformations produce only a subset of all possible query plans
  - only a subset of possible annotations might be considered cost of query plans can only be estimated
- Query Tuning: Make life easier for your query optimizer!

# Figure out problematic queries

- Which queries should be rewritten?
  - Rewrite queries that run "too slow"
  - How to find these queries?
    - query issues far too many disc accesses, for example, point query scans an entire table
    - you look at the query plan and see that relevant indexes are not used

# Overview of query tuning

- avoid DISTINCTs
- subqueries often inefficient
- temporary tables might help
- use clustering indexes for joins
- HAVING vs. WHERE
- use views with care
- system peculiarities: OR and order in FROM clause

# Testbed scenario

- Employee(<u>ssnum,name</u>,manager,dept,salary,numfriends)
  - clustering index on ssnum
  - non-clustering index on name
  - non-clustering index on dept
  - keys: ssnum, name
- Students(<u>ssnum,name</u>,course,grade)
  - clustering index on ssnum
  - non-clustering index on name
  - keys: ssnum, name
- Techdept(<u>dept</u>,manager,location)
  - clustering index on dept
  - key: dept
  - manager may manage many departments
  - a location may contain many departments

# DISTINCT

- How can DISTINCT hurt?
  - DISTINCT forces sort or other overhead.
  - If not necessary, it should be avoided.
- Query: Find employees who work in the information systems department.
  - SELECT DISTINCT ssnum
    FROM Employee
    WHERE dept = 'information systems'
- DISTINCT not necessary:
  - ssnum is a key of Employee, so it is also a key of a subset of Employee.
  - Note: Since an index is defined on ssnum, there is likely to be no overhead in this particular examples.

# Non-Correlated Subqueries

- Many systems handle subqueries inefficiently.
- Non-correlated: attributes of outer query not used in inner query.
- Query:
  - SELECT ssnum
    FROM Employee
    WHERE dept IN (SELECT dept FROM Techdept)
- May lead to inefficient evaluation:
  - check for each employee whether they are in Techdept
  - index on Employee.dept not used!
- Equivalent query:
  - SELECT ssnum
    FROM Employee, Techdept
    WHERE Employee.dept = Techdept.dept
- Efficient evaluation:
  - look up employees for each dept in Techdept
    use index on Employee.dept

# Temporary tables

- Temporary tables can hurt in the following ways:
    - force operations to be performed in suboptimal order (optimizer often does a very good job!)
    - creating temporary tables i.s.s.1 causes catalogue update – possible
      concurrency control bottleneck
    - system may miss opportunity to use index
- Temporary tables are good:
    - to rewrite complicated correlated subqueries
    - to avoid ORDER BYs and scans in specific cases (see example)

# Ex. Unnecessary temp table

- Query: Find all IT department employees who earn more than 40000.
  - SELECT * INTO Temp
    FROM Employee
    WHERE salary > 40000
    SELECT ssnum
    FROM Temp
    WHERE Temp.dept = 'IT'
- Inefficient SQL:
  - index on dept can not be used
  - overhead to create Temp table (materialization vs. pipelining)
- Efficient SQL:
  - SELECT ssnum
    FROM Employee
    WHERE Employee.dept = 'IT'
    AND salary > 40000

# Joins: Use Clustering Indexes and Numeric Values

- Query: Find all students who are also employees.
- Inefficient SQL:
  - SELECT Employee.ssnum
    FROM Employee, Student
    WHERE Employee.name = Student.name
- Efficient SQL:
  - SELECT Employee.ssnum
    FROM Employee, Student
    WHERE Employee.ssnum = Student.ssnum
- Benefits:
  - Join on two clustering indexes allows merge join (fast!).
  - Numerical equality is faster evaluated than string equality.

# Don't use HAVING where WHERE is enough

- Query: Find average salary of the IT department.
- Inefficient SQL:
  - SELECT AVG(salary) as avgsalary, dept
    FROM Employee
    GROUP BY dept
    HAVING dept = 'IT'
- Problem: May first compute average for employees of all departments.
- Efficient SQL: Compute average only for relevant employees.
  - SELECT AVG(salary) as avgsalary, dept
    FROM Employee
    WHERE dept = 'IT'
    GROUP BY dept

# Use views with care

- Views: macros for queries
    - queries look simpler
    - but are never faster and sometimes slower
- Creating a view:
    - CREATE VIEW Techlocation
      AS SELECT ssnum, Techdept.dept, location
      FROM Employee, Techdept
      WHERE Employee.dept = Techdept.dept
- Using the view:
    - SELECT location
      FROM Techlocation
      WHERE ssnum = 452354786
- System expands view and executes:
    - SELECT location
      FROM Employee, Techdept
      WHERE Employee.dept = Techdept.dept
      AND ssnum = 452354786

- Query: Get the department name for the employee with social security number 452354786 (who works in a technical department).
- Example of an inefficient SQL:
  - SELECT dept
    FROM Techlocation
    WHERE ssnum = 452354786
- This SQL expands to:
  - SELECT dept
    FROM Employee, Techdept
    WHERE Employee.dept = Techdept.dept
    AND ssnum = 452354786
- But there is a more efficient SQL (no join!) doing the same thing:
  - SELECT dept
    FROM Employee
    WHERE ssnum = 452354786

# System Peculiarity: Indexes and OR

- Some systems never use indexes when conditions are OR-connected.
- Query: Find employees with name Smith or who are in the acquisitions department.
    - SELECT Employee.ssnum
    FROM Employee
    WHERE Employee.name = 'Smith'
    OR Employee.dept = 'acquisitions'
- Fix: use UNION instead of OR

    - SELECT Employee.ssnum
    FROM Employee
    WHERE Employee.name = 'Smith'
    UNION
    SELECT Employee.ssnum
    FROM Employee
    WHERE Employee.dept = 'acquisitions'