Database Tuning
Query Tuning

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg

database research group
http://dbresearch.uni-salzburg.at

SS 2017/18

Version March 7, 2018

Adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

About Query Tuning

- Query tuning: rewrite a query to run faster!
- Other tuning approaches may have harmful side effects:
  - adding index
  - changing the schema
  - modify transaction length
- Query tuning: only beneficial side effects
  - first thing to do if query is slow!

Outline

Steps in Query Processing

1. Parser
   - input: SQL query
   - output: relational algebra expression
2. Optimizer
   - input: relational algebra expression
   - output: query plan
3. Execution engine
   - input: query plan
   - output: query result

1. Parser

Parser:

- Input: SQL query from user
  Example:
```
SELECT balanace
FROM account
WHERE balance < 2500
```
- Output: relational algebra expression
  Example: $\sigma_{balance<2500}(\Pi_{balance}(account))$
- Algebra expression for a given query not unique!
  Example: The following relational algebra expressions are equivalent.
  - $\sigma_{balance<2500}(\Pi_{balance}(account))$
  - $\Pi_{balance}(\sigma_{balance<2500}(account))$

2. Optimizer

Optimizer:

- Input: relational algebra expression
  Example: $\Pi_{balance}(\sigma_{balance<2500}(account))$
- Output: query plan
  Example:
  $\Pi_{balance}$
  |
  $\sigma_{balance<2500}$
  use index 1
  |
  account
- query plan is selected in three steps:
  A) equivalence transformation
  B) annotation of the relational algebra expression
  C) cost estimation for different query plans

A) 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

Equivalence Rules – Examples

- Selection operations are commutative: $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
  - E is a relation (table)
  - θ_1 and θ_2 are conditions on attributes, e.g. $E.sallary < 2500$
  - σ_θ selects all tuples that satisfy θ
- Selection distributes over the theta-join operation if θ_1 involves only attributes of E_1 and θ_2 only attributes of E_2:

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

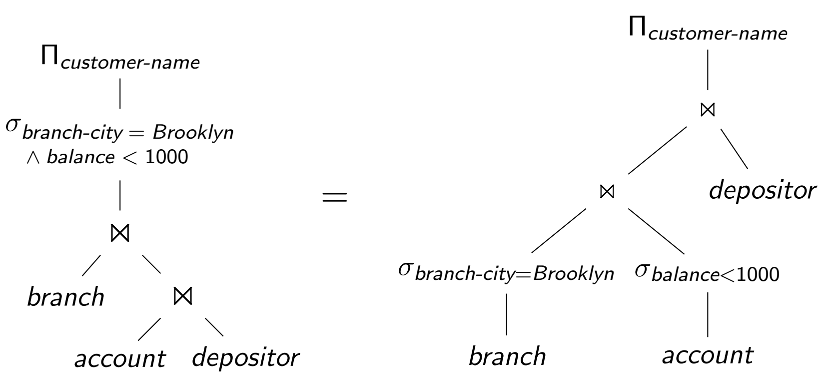
  - $\bowtie_\theta$ is the theta-join; it pairs tuples from the input relations (e.g., E_1 and E_2) that satisfy condition θ, e.g. $E_1.accountID = E_2.ID$
- Natural join is associative: $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  - the join condition in the natural join is equality on all attributes of the two input relations that have the same name
- Many other rules can be found in Silberschatz et al., "Database System Concepts"

Equivalence Rules – Example Query

- Schema:
  branch(branch-name, branch-city, assets)
  account(account-number, branch-name, balance)
  depositor(customer-name,account-number)
- Query:
  SELECT customer-name
  FROM branch, account, depositor
  WHERE branch-city=Brooklyn AND
  balance < 1000 AND
  branch.branch-name = account.branch-name AND
  account.account-number = depositor.account-number

Equivalence Rules – Example Query

- Equivalent relational algebra expressions:

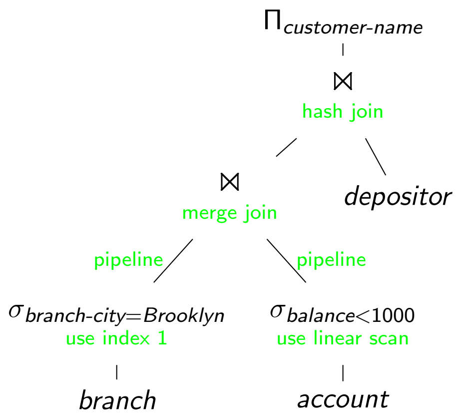


B) Annotation: Creating Query Plans

- 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!

Query Plan – Example

- query plan of our example query:
  (account physically sorted by branch-name; index 1 on branch-city sorts records with same value of branch-city by branch-name)



C) Cost Estimation

- Which query plan is the fastest one?
- This is a very hard problem:
  - cost for each query plan can only be estimated
  - huge number of query plans may exist

Statistics for Cost Estimation

- Catalog information: database maintains statistics about relations
- Example statistics:
  - number of tuples per relation
  - number of blocks on disk per relation
  - number of distinct values per attribute
  - histogram of values per attribute
- Statistics used to estimate cost of operations, for example
  - selection size estimation
  - join size estimation
  - projection size estimation
- 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

3. Execution Engine

The execution engine

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

Query Tuning and Query Optimization

- 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!

Outline

Which Queries Should Be Rewritten?

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

Running Example

- Employee(ssnum,name,manager,dept,salary,numfriends)
  - clustering index on ssnum
  - non-clustering index on name
  - non-clustering index on dept
  - keys: ssnum, name
- Students(ssnum,name,course,grade)
  - clustering index on ssnum
  - non-clustering index on name
  - keys: ssnum, name
- Techdept(dept,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 catalog 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)

[1] in some systems

Unnecessary Temporary 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 (I/II)

- 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
```

Use Views with Care (II/II)

- 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 Techdept.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

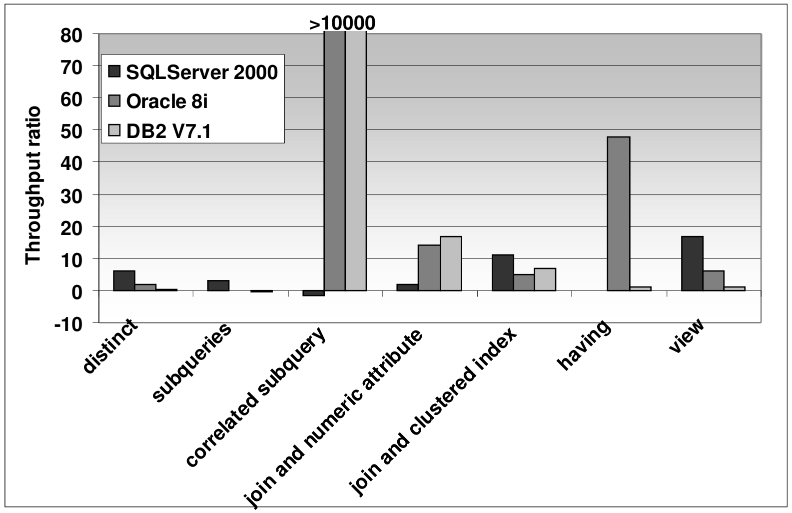
```
SELECT Employee.ssnum
FROM Employee
WHERE Employee.name = 'Smith'
UNION
SELECT Employee.ssnum
FROM Employee
WHERE Employee.dept = 'acquisitions'
```


System Peculiarity: Order in FROM clause

- Order in FROM clause should be irrelevant.
- However: For long joins (e.g., more than 8 tables) and in some systems the order matters.
- How to figure out? Check query plan!

Experimental Evaluation

Throughput increase in percent.



Running Example: 100k employees, 100k students, 10 technical departments

Outline

About Query Tuning

- DISTINCT removes duplicate tuples from the query result.
- Goal: avoid DISTINCT if possible!
- How to know if DISTINCT is necessary?
- We use the notions of
  - privileged tables and
  - reachability

  to decide whether there can be duplicates in the query result.

Privileged Tables

- Privileged table: Attributes returned by SELECT clause contain a key.
- Example: Get the social security numbers of all employees that work in a technical department.

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

- Employee is a privileged table:
  - the SELECT clause projects the attribute ssnum
  - ssnum is a key of Employee

Reachability

- R and S are tables
- R reaches S if
  - R and S are joined on equality and
  - the join attribute in R is a key of R
- Intuition: A tuple from S is joined to at most one tuple from R.
- Reachability is transitive: if A reaches B and B reaches C then A reaches C.

Reachability – Example

- Previous Example: Get the social security numbers of all employees that work in a technical department.

  ```
  SELECT ssnum
  FROM Employee, Techdept
  WHERE Employee.dept = Techdept.dept
  ```
- Techdept reaches Employee:
  - Techdept and Employee are joined on equality
  - dept is a key of Techdept

No-Duplicate Guarantee

- A query returns no duplicates if the following conditions hold:
  - Every attribute in the SELECT clause is from a privileged table.
  - Every unprivileged table reaches at least one privileged one.

No-Duplicate Guarantee – Examples

- This query may return duplicates:

  ```
  SELECT ssnum
  FROM Employee, Techdept
  WHERE Employee.manager = Techdept.manager
  ```
- Reason:
  - manager is not a key of Techdept
  - thus Techdept does not reach privileged table Employee

No-Duplicate Guarantee – Examples

- This query returns no duplicates:

  ```
  SELECT ssnum, Techdept.dept
  FROM Employee, Techdept
  WHERE Employee.manager = Techdept.manager
  ```
- Reason: different from previous example,
  - both Techdept and Employee are privileged table

No-Duplicate Guarantee – Examples

- This query also returns no duplicates:

```
SELECT ssnum, Techdept.dept
FROM Employee, Techdept
```

- Reason: as before,
  - both Techdept and Employee are privileged table

No-Duplicate Guarantee – Examples

- This query returns no duplicates:
  (even if Student.name is not a key)

```
SELECT Student.ssnum
FROM Student, Employee, Techdept
WHERE Student.name = Employee.name
AND Employee.dept = Techdept.dept
```

- Reason:
  - join attribute Employee.name is a key, thus Employee reaches privileged table Student
  - join attribute Techdept.dept is a key thus Techdept reaches Employee
  - transitivity: Techdept reaches Employee and Employee reaches Student, thus Techdept reaches Student

No-Duplicate Guarantee – Examples

- This query returns duplicates:
  (even if Student.name is a key)

```
SELECT Student.ssnum
FROM Student, Employee, Techdept
WHERE Student.name = Employee.name
AND Employee.manager = Techdept.manager
```

- Reason:
  - join attribute Techdept.manager is not key
  - thus Techdept does not reach Employee (and Student)

No-Duplicate Guarantee – Examples

- Try the example queries on the following instance (keys underlined):
  - Employee(<u>ssnum</u>, <u>name</u>, manager, dept)

ssnum	name	manager	dept
1	Peter	John	IT
2	Rose	Mary	Development

  - Techdept(<u>dept</u>, manager)

dept	manager
IT	John
Development	Mary
Production	John

  - Students(<u>ssnum</u>, name)

ssnum	name
5	Peter
6	Peter

Outline

Types of Nested Queries

- Uncorrelated subqueries
  - with aggregates in the inner query

```
SELECT ssnum
FROM Employee
WHERE salary > (SELECT AVG(salary) FROM Employee)
```

  - without aggregates in the inner query

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```


Types of Nested Queries

- Correlated subqueries
  - with aggregates in the inner query

```
SELECT ssnum
FROM Employee e1, Techdept
WHERE salary = (SELECT AVG(e2.salary)
                FROM Employee e2, Techdept
                WHERE e2.dept = e1.dept
                AND e2.dept = Techdept.dept)
```

  - without aggregates in the inner query (uncommon)

Uncorrelated Subquery with Aggregates

- Uncorrelated subqueries with aggregate in the inner query:

```
SELECT ssnum
FROM Employee
WHERE salary > (SELECT AVG(salary) FROM Employee)
```

- Not problematic:
  - Result of inner query is a single value (constant).
  - Most systems will first execute the inner query and then substitute it with the resulting constant.

Uncorrelated Subquery without Aggregates

- Uncorrelated subqueries without aggregate in the inner query:

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

- Some systems might not use index on `Employee.dept`.
- Unnested query:

```
SELECT ssnum
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```


Uncorrelated Subquery without Aggregates

- Unnesting strategy:
  1. Combine the arguments of the two `FROM` clauses.
  2. `AND` together the `WHERE` clauses.
  3. Replace "`outer.attr1 IN (SELECT inner.attr2 ...)`" with "`outer.attr1 = inner.attr2`" in the `WHERE` clause.
  4. Retain the `SELECT` clause from the outer block.
- Strategy works for nesting of any depth.
- Note: If `inner` table does not reach `outer` table in new join condition, new duplicates may appear.

Duplicates in Unnested Queries – Examples

- Nested query:

```
SELECT AVG(salary)
FROM Employee
WHERE dept IN (SELECT dept FROM Techdept)
```

- Unnested query:

```
SELECT AVG(salary)
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
```

- Unnesting is correct:
  - `Techdept` reaches `Employee`, thus no duplicates are introduced
  - each salary appears once in average

Duplicates in Unnested Queries – Examples

- Nested query:

```
SELECT AVG(salary)
FROM Employee
WHERE manager IN (SELECT manager FROM Techdept)
```

- Unnested query:

```
SELECT AVG(salary)
FROM Employee, Techdept
WHERE Employee.manager = Techdept.manager
```

- Unnesting is not correct:
  - `Techdept` does not reach `Employee`, thus duplicates possible
  - some salaries might appears multiple times in the average
- Note: Duplicates do not matter for aggregates like `MIN` and `MAX`.

Duplicates in Unnested Queries – Examples

- Solutions for following query?

```
SELECT AVG(salary)
FROM Employee
WHERE manager IN (SELECT manager FROM Techdept)
```

A) Derived table:

```
SELECT AVG(salary)
FROM Employee, (SELECT DISTINCT manager FROM Techdept) AS T
WHERE Employee.manager = T.manager
```

B) Temporary table:

```
SELECT DISTINCT manager INTO Temp
FROM Techdept

SELECT AVG(salary)
FROM Employee, Temp
WHERE Employee.manager = Temp.manager
```


Correlated Subqueries with Aggregates

- Correlated subquery with aggregates in the inner query:

```
SELECT ssnum
FROM Employee e1, Techdept
WHERE salary = (SELECT AVG(e2.salary)
                FROM Employee e2, Techdept
                WHERE e2.dept = e1.dept
                AND e2.dept = Techdept.dept)
```

- Inefficient in many systems.

Strategy for Rewriting Query

```
SELECT ssnum
FROM Employee e1, Techdept
WHERE salary = (SELECT AVG(e2.salary)
                FROM Employee e2, Techdept
                WHERE e2.dept = e1.dept
                AND e2.dept = Techdept.dept)
```

1. Create temporary table:
   - GROUP BY on correlated attribute of inner query (must be equality!).
   - Use uncorrelated qualifications of inner query for WHERE clause.

```
SELECT AVG(salary) as avsalary, Employee.dept INTO Temp
FROM Employee e2, Techdept
WHERE e2.dept = Techdept.dept
GROUP BY e2.dept
```


Strategy for Rewriting Query

```
SELECT ssnum
FROM Employee e1, Techdept
WHERE salary = (SELECT AVG(e2.salary) ... WHERE e2.dept = e1.dept ...)
```

```
SELECT AVG(salary) as avsalary, Employee.dept INTO Temp
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
GROUP BY Employee.dept
```

2. Join temporary table with outer query:
   - Condition on the grouped attribute replaces correlation condition.
   - Depending attribute of grouping replaces subquery.
   - All other qualifications of outer query remain (none in example).

```
SELECT ssnum
FROM Employee e1, Temp
WHERE salary = avsalary
AND e1.dept = Temp.dept;
```

The Count Bug

- Correlated subquery with COUNT aggregate in the inner query:

```
SELECT ssnum
FROM Employee e1, Techdept
WHERE numfriends = COUNT(SELECT e2.ssnum
                         FROM Employee e2, Techdept
                         WHERE e2.dept = e1.dept
                         AND e2.dept = Techdept.dept)
```

- Rewrite with temporary table:

```
SELECT COUNT(ssnum) as numcolleagues, Employee.dept INTO Temp
FROM Employee, Techdept
WHERE Employee.dept = Techdept.dept
GROUP BY Employee.dept
```

```
SELECT ssnum
FROM Employee, Temp
WHERE numfriends = numcolleagues
AND Employee.dept = Temp.dept;
```

- What is going wrong?

The Count Bug

- Consider for example an employee Jane:
  - Jane is not in a technical department (`Techdept`).
  - Jane has no friends (`Employee.numfriends = 0`)
- Original (nested) query:
  - since Jane is not in a technical department, inner query is empty
  - but COUNT(∅)=0, thus Jane is in the result set!
- Rewritten query with temporary table:
  - Jane not in a technical department and does not survive the join
  - thus Jane is not in the result set