PA152: Efficient Use of DB

9. Query Tuning

Vlastislav Dohnal

Credits

- Sources of materials for this lecture:
  - Courses CS245, CS345, CS345
    - Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom
    - Stanford University, California
  - Database Tuning (slides)
    - Dennis Shasha, Philippe Bonnet
    - Morgan Kaufmann, 1st edition, 440 pages, 2002
    - ISBN-13: 978-1558607538
    - http://www.databasetuning.org/

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.
       FROM TABLES_AVAILABLE t, SUPPLIER_INFO
       WHERE t.SUPPLIER_ID = s.SUPPLIER_ID
             and (TO_CHAR(t.DATE_TIME, 'MM/DD/YY
              or TO_NUMBER(TO_CHAR(sysdate, 'SSSS
             and t.NUM_OFFERS  > 0 and t.DATE_TIM
             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 > 2 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 the query evaluated?
2) How can we speed it up?

Query Execution Plan

Output of EXPLAIN command in Oracle

```
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)
```

Operator

Access method

Evaluation cost

Monitoring Queries

- What is slow query?
  - Needs to many disk IOs
    - high *costs* in execution plan (explain)
    - E.g., query for one row (exact-match query) uses table-scan.
  - Inconvenient query plan
    - Existing indexes are not used
- How to reveal?
  - DBMS can log "long-lasting" queries
  - …

Query Tuning

- Local tuning = query rewrite
  - First approach to speed up a query
  - Influences only the query
- Global tuning
  - Index creation
  - Schema modification
  - Transaction splitting
  - …
  - Potentially harmful

Query Rewriting

- Example:
  - Employee(<u>ssnum</u>, name, manager, dept, salary, coworkers)
    - Clustering index on *ssnum*
      - i.e., relation is sorted by this attribute in the file
    - Non-clustering indexes: (i) *name;* (ii) *dept*
  - Student(<u>ssnum</u>, name, degree_sought, year)
    - Clustering index on *ssnum*
    - Non-clustering index on *name*
  - Tech(<u>dept</u>, manager, location)
    - Clustering index on *dept*

Query Rewriting

- Techniques
  - Index usage
  - DISTINCTs elimination
  - (Correlated) subqueries
  - Use of temporaries
  - Use of having
  - Use of views
  - Materialized views

Index Usage

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

    WHERE salary/12 >= 4000;

    WHERE inserted + 1 = current_date;
  - Functions

    SELECT * FROM employee
      WHERE SUBSTR(name, 1, 1) = ‘G’;

    … WHERE to_char(inserted, 'YYYYMM') = '201704'
  - Numerical comparisons of fields with different types
  - Multi-attribute indexes
  - Comparison with NULL

Index Usage

- = vs. like
  - SELECT * FROM hotel WHERE city='city174'
  - SELECT * FROM hotel WHERE city LIKE 'city174'

```
"Bitmap Heap Scan on hotel  (cost=4.31..14.26 rows=5 width=59)"
"  Filter: ((city)::text ~~ 'city174'::text)"
"  ->  Bitmap Index Scan on hotel_city  (cost=0.00..4.31 rows=5 width=0)"
"        Index Cond: ((city)::text = 'city174'::text)"
```

  - SELECT * FROM hotel WHERE city like 'city174%'

```
"Seq Scan on hotel  (cost=0.00..17.25 rows=5 width=59)"
"  Filter: ((city)::text ~~ 'city174%'::text)"
```

Index Usage (cont.)

- Aggregate functions MAX(*A*), MIN(*A*)
  - resp. ORDER BY *A* LIMIT 1
  - using functions on A
  - E.g.,
    - conn_log (log_key, sim_imsi, time, car_key, pda_imei, gsmnet_id, method, program_ver)

Plus a secondary index on (sim_imsi,time)

A. SELECT **max(time AT TIME ZONE 'UTC')** AS time
FROM conn_log
WHERE sim_imsi='23001234567890123' AND
time>'2016-02-28 10:50:00.122 UTC' AND
method='U' AND program_ver IS NOT NULL;

B. SELECT **time AT TIME ZONE 'UTC'**
FROM (SELECT **max(time)** AS time
FROM conn_log
WHERE sim_imsi='23001234567890123' AND
time>'2016-02-28 10:50:00.122 UTC' AND
method='U' AND program_ver IS NOT NULL) AS x;

C. SELECT **max(time) AT TIME ZONE 'UTC'** AS time …
(cont. from A.)

Index Usage (cont.)

```
QUERY PLAN  (QUERY A.)

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=19412.69..19412.70 rows=1 width=8) (actual time=36.415..36.415 rows=1 loops=1)
  ->  Append  (cost=0.00..19385.45 rows=5448 width=8) (actual time=36.410..36.410 rows=0 loops=1)
        ->  Seq Scan on conn_log  (cost=0.00..0.00 rows=1 width=8) (actual time=0.003..0.003 rows=0 loops=1)
              Filter: ((program_ver IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone) AND (sim_imsi = '23001234567890123'::bpchar) AND
                                                                                                                                                        (method = 'U'::bpchar))
        ->  Index Scan using conn_log_imsi_time_y2016m02 on conn_log_y2016m02  (cost=0.56..8.58 rows=1 width=8) (actual time=28.464..28.464 rows=0 loops=1)
              Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
              Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
        ->  Bitmap Heap Scan on conn_log_y2016m03  (cost=194.11..14125.36 rows=3969 width=8) (actual time=2.586..2.586 rows=0 loops=1)
              Recheck Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
              Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
              ->  Bitmap Index Scan on conn_log_imsi_time_y2016m03  (cost=0.00..193.12 rows=4056 width=0) (actual time=2.584..2.584 rows=0 loops=1)
                    Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
        ->  Bitmap Heap Scan on conn_log_y2016m04  (cost=71.87..5243.35 rows=1476 width=8) (actual time=5.346..5.346 rows=0 loops=1)
              Recheck Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
              Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
              ->  Bitmap Index Scan on conn_log_imsi_time_y2016m04  (cost=0.00..71.50 rows=1507 width=0) (actual time=5.342..5.342 rows=0 loops=1)
                    Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
        ->  Index Scan using conn_log_imsi_time_y2016m05 on conn_log_y2016m05  (cost=0.14..8.16 rows=1 width=8) (actual time=0.009..0.009 rows=0 loops=1)
              Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
              Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
Planning time: 4.159 ms
Execution time: 36.535 ms
```

Index Usage (cont.)

```
QUERY PLAN  (QUERY B.)

---------------------------------------------------------------------------------------------------------------------------------------------------------------
Subquery Scan on x  (cost=5.98..6.01 rows=1 width=8) (actual time=0.162..0.163 rows=1 loops=1)
  ->  Result  (cost=5.98..5.99 rows=1 width=0) (actual time=0.159..0.160 rows=1 loops=1)
        InitPlan 1 (returns $0)
          ->  Limit  (cost=1.87..5.98 rows=1 width=8) (actual time=0.158..0.158 rows=0 loops=1)
                ->  Merge Append  (cost=1.87..22424.61 rows=5449 width=8) (actual time=0.156..0.156 rows=0 loops=1)
                      Sort Key: conn_log."time"
                      ->  Index Scan Backward using conn_log_imsi_time on conn_log  (cost=0.12..8.15 rows=1 width=8) (actual time=0.004..0.004 rows=0 loops=1)
                            Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                            Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                      ->  Index Scan Backward using conn_log_imsi_time_y2016m02 on conn_log_y2016m02  (cost=0.56..8.58 rows=1 width=8)
                                                                                                                    (actual time=0.069..0.069 rows=0 loops=1)
                            Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                            Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                      ->  Index Scan Backward using conn_log_imsi_time_y2016m03 on conn_log_y2016m03  (cost=0.56..16225.91 rows=3969 width=8)
                                                                                                                    (actual time=0.046..0.046 rows=0 loops=1)
                            Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                            Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                      ->  Index Scan Backward using conn_log_imsi_time_y2016m04 on conn_log_y2016m04  (cost=0.43..6033.60 rows=1477 width=8)
                                                                                                                    (actual time=0.035..0.035 rows=0 loops=1)
                            Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                            Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                      ->  Index Scan Backward using conn_log_imsi_time_y2016m05 on conn_log_y2016m05  (cost=0.14..8.17 rows=1 width=8)
                                                                                                                    (actual time=0.002..0.002 rows=0 loops=1)
                            Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                            Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
Planning time: 3.137 ms
Execution time: 0.317 ms
```

Index Usage (cont.)

```
 QUERY PLAN  (QUERY C.)

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Result  (cost=5.98..5.99 rows=1 width=0) (actual time=0.186..0.186 rows=1 loops=1)
  InitPlan 1 (returns $0)
    ->  Limit  (cost=1.87..5.98 rows=1 width=8) (actual time=0.182..0.182 rows=0 loops=1)
          ->  Merge Append  (cost=1.87..22424.63 rows=5450 width=8) (actual time=0.181..0.181 rows=0 loops=1)
                Sort Key: conn_log."time"
                ->  Index Scan Backward using conn_log_imsi_time on conn_log  (cost=0.12..8.15 rows=1 width=8) (actual time=0.005..0.005 rows=0 loops=1)
                      Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                      Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                ->  Index Scan Backward using conn_log_imsi_time_y2016m02 on conn_log_y2016m02  (cost=0.56..8.58 rows=1 width=8)
                                                                                                  (actual time=0.070..0.070 rows=0 loops=1)
                      Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                      Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                ->  Index Scan Backward using conn_log_imsi_time_y2016m03 on conn_log_y2016m03  (cost=0.56..16225.91 rows=3969 width=8)
                                                                                                  (actual time=0.064..0.064 rows=0 loops=1)
                      Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                      Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                ->  Index Scan Backward using conn_log_imsi_time_y2016m04 on conn_log_y2016m04  (cost=0.43..6033.60 rows=1478 width=8)
                                                                                                  (actual time=0.037..0.037 rows=0 loops=1)
                      Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                      Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
                ->  Index Scan Backward using conn_log_imsi_time_y2016m05 on conn_log_y2016m05  (cost=0.14..8.17 rows=1 width=8)
                                                                                                  (actual time=0.003..0.003 rows=0 loops=1)
                      Index Cond: ((sim_imsi = '23001234567890123'::bpchar) AND ("time" IS NOT NULL) AND ("time" > '2016-02-28 11:50:00.122+01'::timestamp with time zone))
                      Filter: ((program_ver IS NOT NULL) AND (method = 'U'::bpchar))
 Planning time: 3.094 ms
 Execution time: 0.309 ms
```

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 is unnecessary
  - *ssnum* is a prim. key in *employee*

Example of DISTINCTs

- Assume the relation hotel in student's Pg

explain select **distinct** id from hotel where id is not null;
"Unique (cost=0.00..**33.00** rows=500 width=4)"
" -> Index Scan using hotel_pkey on hotel (cost=0.00..31.75 rows=500 width=4)"
"   Filter: (id IS NOT NULL)"

explain select id from hotel where id is not null;
"Seq Scan on hotel (cost=0.00..**10.00** rows=500 width=4)"
" Filter: (id IS NOT NULL)"

explain select **distinct** id from account where id < 1000;
"Unique (cost=0.00..**62.13** rows=993 width=4)"
" -> Index Scan using account_pkey on account (cost=0.00..59.65 rows=993 width=4)"
"   Index Cond: (id < 1000)"

explain select id from account where id < 1000;
"Index Scan using account_pkey on account (cost=0.00..**59.65** rows=993 width=4)"
" Index Cond: (id < 1000)"

Eliminate unneeded DISTINCTs

- Query:

  Employee(<u>ssnum</u>, name, manager, dept, salary, coworkers)
  Tech(<u>dept</u>, manager, location)

  - 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?

Eliminate unneeded DISTINCTs

Employee(<u>ssnum</u>, name, manager, dept, salary, coworkers)
Tech(<u>dept</u>, manager, location)

- Query:
  - SELECT DISTINCT ssnum
    FROM employee, tech
    WHERE employee.dept = tech.dept
- Is DISTINCT needed?
  - *ssnum* is a key in *employee*
  - *dept* is a key in *tech*
  - → each employee record will join with at most one record in *tech.*
  - → DISTINCT is unnecessary

Eliminate unneeded DISTINCTs

- The relationship among DISTINCT, keys and joins can be generalized:
  - Definition of *“privileged*“
    - Call a table T *privileged* if the fields returned by the select contain a key of T.
  - Definition of relationship *“reaches*“
    - 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.
  - Relationship “*reaches“* is transitive:
    - If R_1 reaches R_2 and R_2 reaches R_3, then R_1 reaches R_3.

Eliminate unneeded DISTINCTs

- 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*.

Unneeded DISTINCT (1)

Employee(<u>ssnum</u>, name, manager, dept, salary, coworkers)
Tech(<u>dept</u>, manager, location)

- Query:
  - SELECT DISTINCT ssnum
    FROM employee, tech
    WHERE employee.manager = tech.manager
- *Employee* is privileged
- Is *tech* privileged?
  - No.
- Does *tech* reach *employee*?
  - No. Attribute *manager* is not a key in *tech*.

Unneeded DISTINCT (2)

Employee(ssnum, name, manager, dept, salary, coworkers)
Tech(dept, manager, location)

- Query:
  - SELECT DISTINCT ssnum, tech.dept
    FROM employee, tech
    WHERE employee.manager = tech.manager
- *Employee* is privileged
- Is *tech* privileged?
  - Yes.
- Result does not have duplicates

Unneeded DISTINCT (3)

- Query:

Employee(<u>ssnum</u>, name, manager, dept, salary, coworkers)
Student(<u>ssnum</u>, name, degree_sought, year)
Tech(<u>dept</u>, manager, location)

  - SELECT DISTINCT student.ssnum
    FROM student, employee, tech
    WHERE student.name = employee.name
    AND employee.dept = tech.dept;
- *Student* is privileged
- *Employee* is not privileged and does not reach any other relation.
- → DISTINCT is needed.

Nested Queries

- SELECT containing another SELECT as its part
  - SELECT employee_number, name
    FROM employees AS X
    WHERE salary > (
      SELECT AVG(salary)
      FROM employees
      WHERE department = X.department);

  - SELECT employee_number, name,
      (*SELECT AVG(salary) FROM employees*
       WHERE department = X.department) AS department_average
    FROM employees AS X;

Rewriting Nested Queries

- Reason:
  - Query optimizer may not correctly handle some nested queries
  - Usually:
    - Uncorrelated subqueries without aggregate
    - Correlated subqueries

Types of Nested Queries

- Uncorrelated subqueries with aggregates
  SELECT ssnum FROM employee
  WHERE salary >
  (SELECT avg(salary) FROM employee)
- Uncorrelated subqueries without aggregate
  SELECT ssnum FROM employee
  WHERE dept in (SELECT dept FROM tech)
  - So-called “semi-join”

Types of Nested Queries

- Correlated subqueries with aggregates
  - ```
    SELECT ssnum FROM employee e1
    WHERE salary >=
      (SELECT avg(e2.salary)
        FROM employee e2, tech
        WHERE e2.dept = e1.dept
            AND e2.dept = tech.dept)
    ```
    ```

Types of Nested Queries

- Correlated subqueries without aggregates
  - Unusual for derived tables
    - i.e., can rewrite with join
  - Subqueries in where (typical)
    - Semi-join queries may be evaluated efficiently
    - Example of two semi-join queries:
      - SELECT ssnum FROM employee
        WHERE dept in
        (SELECT dept FROM tech
        WHERE tech.manager=employee.manager)
      - SELECT ssnum FROM employee
        WHERE EXISTS (SELECT 1 FROM tech WHERE
        employee.manager = tech.manager)

Rewriting Uncorrel. Subq. without Aggregates

1. Combine the arguments of the two FROM clauses
2. Replace IN with =
3. Retain the SELECT clause

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

↓

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

Rewriting Uncorrel. Subq. without Aggregates

- Potential problem with duplicates:
  - SELECT avg(salary) FROM employee
    WHERE manager in (select manager from tech)
  - SELECT avg(salary) FROM employee, tech
    WHERE employee.manager = tech.manager
- The rewritten query may include an employee record several times
  - if that employee's manager manages several departments.
- The solution is to create a temporary table
  - (using DISTINCT) to eliminate duplicates.

Rewriting Correlated Subqueries

- Query:
  - Find the employees of tech departments who earn at least the average salary in their department.

```
SELECT ssnum
FROM employee e1
    WHERE salary >= (SELECT avg(e2.salary)
                     FROM employee e2, tech
                     WHERE e2.dept = tech.dept
                       AND e2.dept = e1.dept);
```

Rewriting Correlated Subqueries

```
CREATE TEMPORARY TABLE temp ( … ) ON COMMIT DROP;

INSERT INTO temp
        SELECT avg(salary) as avsalary, tech.dept
        FROM tech, employee
        WHERE tech.dept = employee.dept
        GROUP BY tech.dept;

SELECT ssnum
FROM employee, temp
WHERE salary >= avsalary
   AND employee.dept = temp.dept
```

Rewriting Correlated Subqueries

```
SELECT ssnum
FROM employee as E,
        (SELECT avg(salary) as avsalary, tech.dept
         FROM tech, employee
         WHERE tech.dept = employee.dept
         GROUP BY tech.dept) as AVG
WHERE salary >= avsalary AND E.dept = AVG.dept
```

Rewriting Correlated Subqueries

- Query:
  - Find employees of technical departments whose number of co-workers equals the number of employees in their department.

```
SELECT ssnum
FROM employee e1
WHERE coworkers = (
                SELECT COUNT(e2.ssnum)
                FROM employee e2, tech
                WHERE e2.dept = tech.dept
                      AND e2.dept = e1.dept);
```

Rewriting Correlated Subqueries

```
INSERT INTO temp
        SELECT COUNT(ssnum) as numworkers,
                employee.dept
        FROM tech, employee
        WHERE tech.dept = employee.dept
        GROUP BY tech.dept;

SELECT ssnum
        FROM employee, temp
        WHERE coworkers = numworkers
                AND employee.dept = temp.dept;
```

Can you spot the infamous COUNT bug?

The Infamous COUNT Bug

- Example:
  - Helene who is not in a technical department.
  - In the original query, Helene's number of coworkers would be compared to COUNT(Ø)=0.
    - In case Helene has no coworkers, she would survive the selection.
  - In the transformed query, Helene's record would not appear.
    - The temporary table will contain counts for tech departments only.
- This is a limitation of the correlated subquery rewriting technique when COUNT is involved.

Rewriting Correlated Subqueries

- Anti-joins
  - SELECT * FROM Tech WHERE dept NOT IN (SELECT dept FROM employee)
    - Problem with NULLs in employee.dept
  - SELECT * FROM Tech WHERE NOT EXISTS (SELECT 1 FROM employee WHERE employee.dept=tech.dept)
- Issues
  - Not using join algorithm
  - Using too many index lookups in index join

Rewriting Correlated Subqueries

- Test these in student's Pg:

```
explain verbose select * from hotel
  where id not in (select hotel_id from room);
"Seq Scan on xdohnal.hotel  (cost=0.00..2190904.75 rows=250 width=59)“
"  Output: hotel.id, hotel.name, hotel.street, …“
"  Filter: (NOT (SubPlan 1))“
"  SubPlan 1“
"    ->  Materialize  (cost=0.00..7974.90 rows=315460 width=4)“
"          Output: room.hotel_id“
"          ->  Seq Scan on xdohnal.room  (cost=0.00..5164.60 rows=315460 width=4)“
"                Output: room.hotel_id"

explain verbose select * from hotel
  where id not in (select hotel_id from room
                   where hotel_id is not null);

explain verbose select * from hotel
  where not exists(select 1 from room
                   where room.hotel_id=hotel.id);
```

Query Rewriting

- Techniques
  - Index usage
  - DISTINCTs elimination
  - (Correlated) subqueries
  - **Use of temporaries**
  - Use of having
  - Use of views
  - Materialized views

Abuse of Temporaries

- Query:
  - Find all information about department employees with their locations who earn at least > 40000.
  - INSERT INTO temp
    SELECT *
    FROM employee
    WHERE salary >= 40000
  - SELECT ssnum, location
    FROM temp
    WHERE temp.dept = ‘information systems’
- This solution will not be optimal (should have been done in the reverse order)
  - Cannot use on *dept* in *employee*
  - There is no index on *temp* table.

Use of Having

- Reason for having:
  - Shortens queries that filter on aggregation results
  - Cannot use aggregations in WHERE clause
  - Use HAVING clause then
- Example
  - SELECT avg(salary), dept
    FROM employee
    GROUP BY dept
    HAVING avg(salary) > 10 000;

Use of Having

- Another example

```
SELECT avg(salary), dept
FROM employee
GROUP BY dept
HAVING count(ssnum) > 100;
```

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;~~
```

Use of Views

```
CREATE VIEW techlocation AS
    SELECT ssnum, tech.dept, location
    FROM employee, tech
    WHERE employee.dept = tech.dept;

SELECT location
FROM techlocation
WHERE ssnum = 43253265;
```

- Query optimizer replaces the view with its definition

Use of Views

- Resulting query:

```
SELECT location
FROM employee, tech
WHERE employee.dept = tech.dept
      AND ssnum = 43253265;
```

Use of Views

- Example for PostgreSQL:
  - CREATE VIEW hotels_in_city AS
    SELECT city, COUNT(*) AS count
    FROM hotel
    GROUP BY city;
- Using view
  - SELECT * FROM hotels_in_city
    WHERE count > 8
  - SELECT * FROM hotels_in_city
    WHERE city='city174'

Use of Views

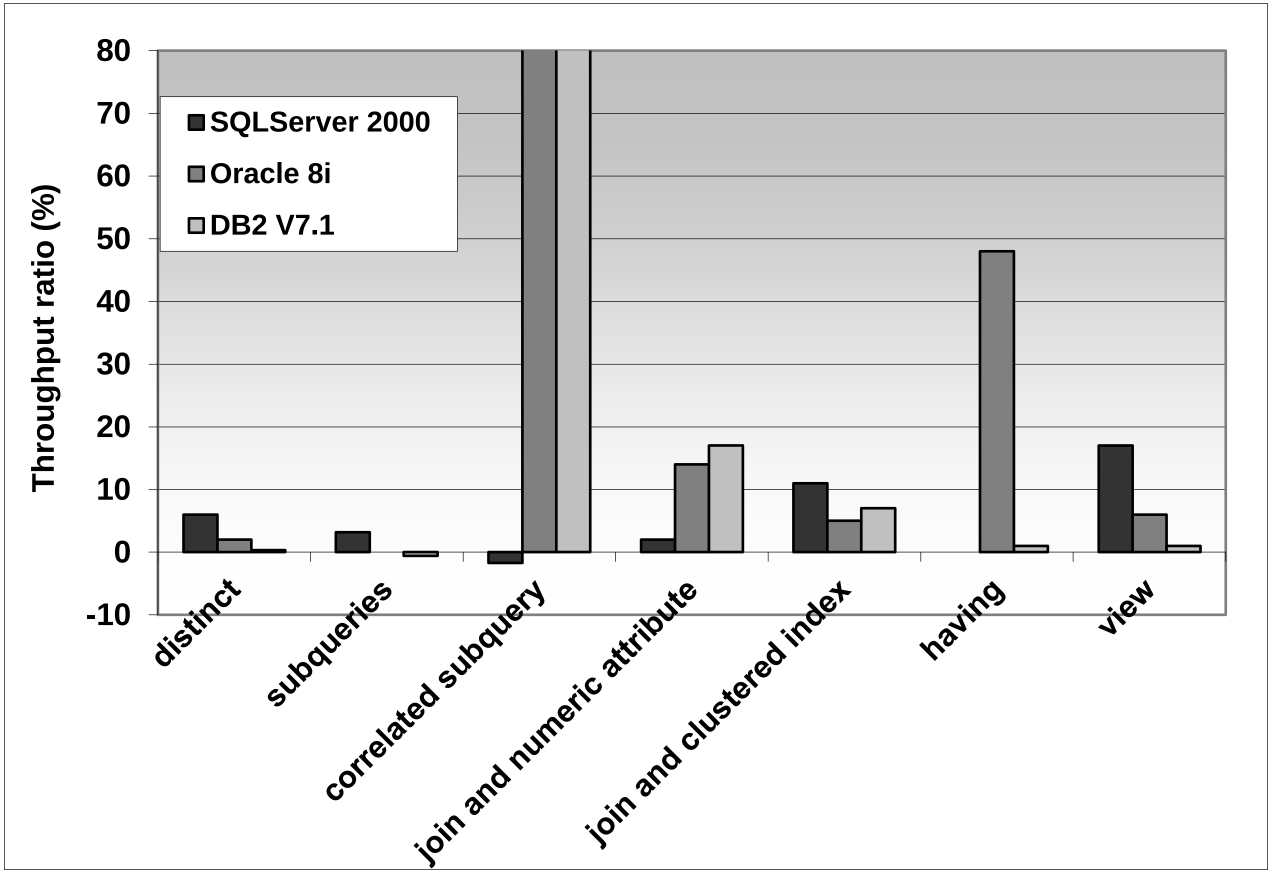
- Output of EXPLAIN
  - EXPLAIN SELECT * FROM hotels_in_city;
  - EXPLAIN SELECT * FROM hotels_in_city WHERE city='city174';
  - Use of functions:
    - Compare:

```
EXPLAIN SELECT * FROM
    (SELECT lower(city) as city, COUNT(*) AS cnt
     FROM hotel GROUP BY city HAVING COUNT(*) > 3) x
  WHERE city='city174';

EXPLAIN SELECT lower(city), cnt FROM
    (SELECT city, COUNT(*) AS cnt FROM hotel
     GROUP BY city HAVING COUNT(*) > 3) x
  WHERE city='city174';
```

Query Rewriting: Performance Impact

>10 000



100k Employees, 100k Students, 10 tech. depts

Aggregate Maintenance

- Example:
  - Orders of a store chain
    - Order(ordernum, itemnum, quantity, purchaser, vendor)
    - Item(itemnum, description, price)
    - Clustered indexes on *itemnum* of *Order* and *Item*
  - Queries issues every five minutes :
    - The total dollar amount of orders from a particular vendor.
    - The total dollar amount of orders by a particular store outlet (purchaser).

Aggregate Maintenance

- Queries:
  - SELECT vendor, sum(quantity*price)
    FROM order, item
    WHERE order.itemnum = item.itemnum
    GROUP BY vendor;
  - SELECT purchaser, sum(quantity*price)
    FROM order, item
    WHERE order.itemnum = item.itemnum
    GROUP BY purchaser;
- Query costs?
  - $\rightarrow$ expensive

Aggregate Maintenance

- Ways to speed up?
  - Use of views?
    - $\rightarrow$ no impact
  - Use of temporaries?
    - $\rightarrow$ helps

Aggregate Maintenance

- Add temporaries
  - OrdersByVendor(<u>vendor</u>, amount)
  - OrdersByPurchaser(<u>purchaser</u>, amount)
- These redundant tables must be updated
  - When to update?
    - After each update to *order*, or *item*?
      - triggers can be used to implement this explicitly
    - Recreate from scratch periodically
  - Costs of update
    - Update overhead must be less than original costs.

Materialized Views

- View data content stored in a table
  - Automatic updates by DBMS
    - Typical…
  - Transparent expansion performed by the optimizer based on cost
    - It is the optimizer and not the programmer that performs query rewriting

Materialized Views

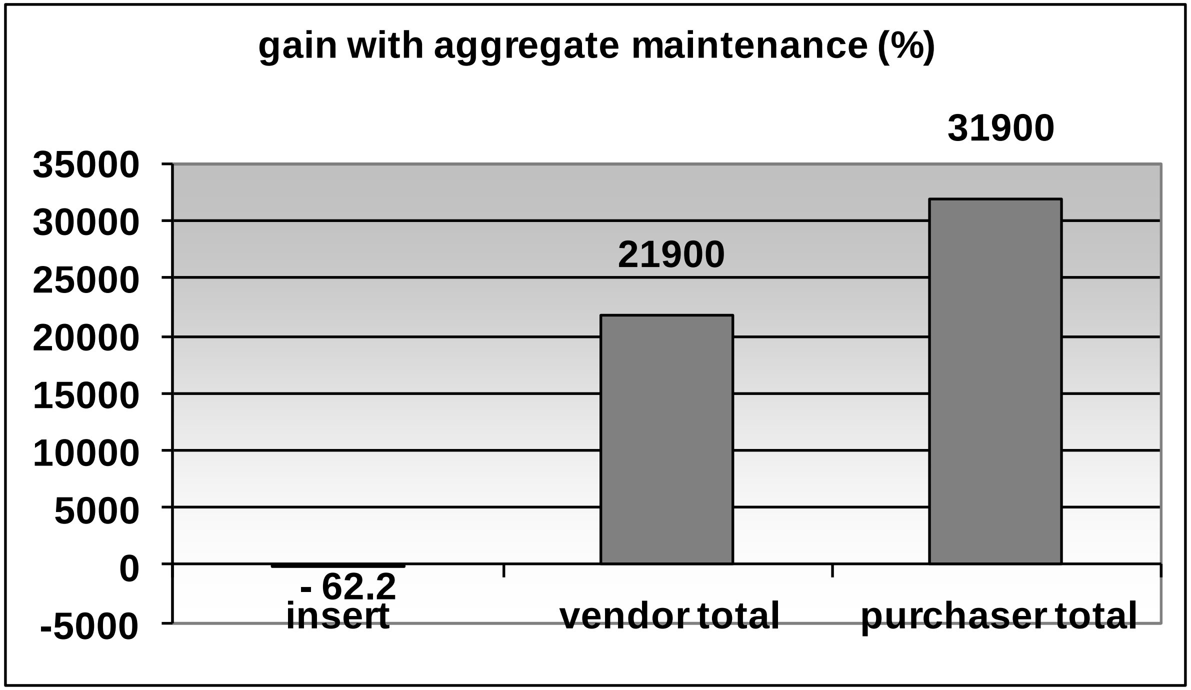
- In Oracle
  - CREATE MATERIALIZED VIEW OrdersByVendor
    BUILD IMMEDIATE REFRESH COMPLETE
    ENABLE QUERY REWRITE
    AS
    SELECT vendor, sum(quantity*price) AS amount
    FROM order, item
    WHERE order.itemnum = item.itemnum
    GROUP BY vendor;

Materialized Views

- Example
  - QUERY REWRITE
  - Query:
    - SELECT vendor, sum(quantity*price) AS amount
      FROM order, item
      WHERE order.itemnum = item.itemnum
      AND vendor=‘Apple’;
    - OrdersByVendor view will be substituted:
      - SELECT vendor, amount FROM OrdersByVendor
        WHERE vendor=‘Apple’;

Materialized Views

- Example
  - SQLServer, using triggers for maintenance
  - 1m orders – 5 purchasers and 20 vendors
  - 10k items



Database Triggers

- A trigger is a stored procedure
  - Collection of SQL statements that executes as a result of an event.
- Events:
  - DML – insert, update, delete
  - DDL – definition of tables, …
  - Time-related events (not common)

Database Triggers

- Independent of an application/API
  - Executed as part of the transaction containing the enabling event by DBMS.
- Not using triggers requires implementation of constraints in app
- Induce overhead
  - May insert to other tables, …
  - Firing can be conditional
    - E.g., after price update, number of ordered items
    - Not on updates to item description, …

Global (Schema) Changes

- Materialized views
  - If refreshed automatically…
- Creating indexes
- Schema change
  - See the next slides
- Relation partitioning
  - See the next slides
- …

Using Indexes

- Small table
  - Indexes created
  - But not used
- Example
  - courses(<u>id</u>, title, credits)
  - SELECT COUNT(*) FROM courses;
    - Result: 5
  - SELECT * FROM courses WHERE id='MA102';
    - Table-scan is used

Using Indexes

- Relation read sequentially (table scan / seq scan)
  - All records are checked
  - $\rightarrow$ slow
- Creating index (index scan)
  - Speeds up SELECTs
  - Slows down INSERTs, UPDATEs, DELETEs
    - Indexes must be updated

Influence of Indexes on Costs

- False friends
  - More indexes, faster evaluation!
    - In theory, valid only for SELECT queries
- Each index increases update costs
  - Necessary to update both relation and index
  - Exception:
    - INSERT INTO table SELECT …
    - DELETE FROM table WHERE …

Influence of Indexes: Example

- Relation
  - StarsIn(id, movieTitle, movieYear, starName)
- Q_{movies}
  - SELECT movieTitle, movieYear FROM StarsIn
    WHERE starName='name';
- Q_{stars}
  - SELECT starName FROM StarsIn
    WHERE movieTitle='title' AND movieYear=year;
- Insert
  - INSERT INTO StarsIn (movieTitle, movieYear,starName)
    VALUES ('title', year, 'name');

Influence of Indexes: Example

- Assumptions
  - B(StarsIn) = 10 blocks
  - Each actor stars in 3 movies on average
  - Each movie has 3 stars on average
  - Relation is not sorted
    - If index is present, 3 reads of disk (3 records).
  - Searching in index
    - 1 block read
  - Index update
    - 1 block read and 1 block write
  - Insert to relation
    - 1 block read and 1 block write
      - i.e., not locating any free block

Influence of Indexes: Example

- Costs in blocks for individual operations
  - Probability of individual operations
    - $Q_{movies}=p_1$, $Q_{stars}=p_2$, Insert=1 - p_1 - p_2

Operation	No indexes	Index *starName*	Index *movieTitle, movieYear*	Both indexes
Q_{movies}	10	4	10	4
Q_{stars}	10	10	4	4
Insert	2	4	4	6
Avg. costs	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

- Scenario 1: $p_1 = p_2 = 0.1 \rightarrow$ no indexes
- Scenario 2: $p_1 = p_2 = 0.4 \rightarrow$ both indexes

Optimizing Indexes

1. Define a batch of operations
   - i.e., composition of load
   - Analyze log files to find out query types, updates and their frequencies
2. Suggest different indexes
   - Optimizer estimates costs to evaluate the batch
   - Choose a configuration with least costs
   - Create corresponding indexes

Optimizing Indexes

- Point 2 in detail:
  - A set of possible indexes
  - Initially without any index
  - Repeat
    - Estimate costs of batch for each possible index
    - Create the index offering the greatest decrease of costs
      - Use it in next iterations
    - Repeat until an index has been created

- The process can be done automatically
  - MS AutoAdmin (http://research.microsoft.com/en-us/projects/autoadmin/default.aspx)
    - MS Index Tuning Wizard (S. Chaudhuri, V. Narasayya: *An efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server*. Proceedings of VLDB Conference, 1997) & the best 10-year paper in 2007!
  - Oracle 10g (http://www.oracle-base.com/articles/10g/AutomaticSQLTuning10g.php)

Referential Integrity

- Creating foreign key may not induce an index on the key’s attributes
- Example in PostgreSQL (db.fi.muni.cz)
  - Hotel – primary key *id*
  - Room – primary key *id*, foreign key *hotel_id*
    - V(Room, hotel_id) = 6
- Queries (check EXPLAIN plans)

```
SELECT * FROM hotel WHERE id=2;
SELECT * FROM room WHERE hotel_id=2 AND number=1;
```

Referential Integrity

- Query

```
SELECT * FROM room WHERE hotel_id=2 AND number=1;
```

- No indexes (output of EXPLAIN SELECT…)

```
Seq Scan on room  (cost=0.00..8750.89 rows=105 width=22)
  Filter: ((hotel_id = 2) AND (number = 1))
```

- Create an index on *hotel_id*

```
CREATE INDEX room_hotel_id_fkey ON room (hotel_id);

Bitmap Heap Scan on room  (cost=974.87..5782.99 rows=105 width=22)
  Recheck Cond: (hotel_id = 2)
  Filter: (number = 1)
  ->  Bitmap Index Scan on room_hotel_id_fkey  (cost=0.00..974.84 rows=52608 width=0)
        Index Cond: (hotel_id = 2)
```

Referential Integrity

- Foreign keys may slow down deletions drastically
- Example
  - DELETE FROM hotel WHERE id=500;
    - Foreign key in *room* references table *hotel*
    - During deletion *room* must be checked for existence of records *hotel_id=500*
- Recommendation
  - Create indexes on foreign keys

Combining Indexes

- Query SELECT * FROM room WHERE hotel_id=2 AND number=1;
- Index only on *hotel_id*

```
"Bitmap Heap Scan on room  (cost=960.80..5756.77 rows=103 width=22)"
"  Recheck Cond: (hotel_id = 2)"
"  Filter: (number = 1)"
"  ->  Bitmap Index Scan on room_hotel_id_fkey  (cost=0.00..960.77 rows=51798 width=0)"
"        Index Cond: (hotel_id = 2)"
```

- Index only on *number*

```
"Bitmap Heap Scan on room  (cost=13.02..1688.30 rows=103 width=22)"
"  Recheck Cond: (number = 1)"
"  Filter: (hotel_id = 2)"
"  ->  Bitmap Index Scan on room_number_idx  (cost=0.00..12.99 rows=628 width=0)"
"        Index Cond: (number = 1)"
```

Combining Indexes

- Query SELECT * FROM room WHERE hotel_id=2 AND number=1;
- Index on *hotel_id, number*

```
"Bitmap Heap Scan on room  (cost=5.34..366.14 rows=103 width=22)"
"  Recheck Cond: ((hotel_id = 2) AND (number = 1))"
"  ->  Bitmap Index Scan on room_hotel_id_number_fkey  (cost=0.00..5.31 rows=103 width=0)"
"        Index Cond: ((hotel_id = 2) AND (number = 1))"
```

- Two indexes on *hotel_id* and *number*

```
"Bitmap Heap Scan on room  (cost=974.07..1334.86 rows=103 width=22)"
"  Recheck Cond: ((number = 1) AND (hotel_id = 2))"
"  ->  BitmapAnd  (cost=974.07..974.07 rows=103 width=0)"
"        ->  Bitmap Index Scan on room_number_idx  (cost=0.00..12.99 rows=628 width=0)"
"              Index Cond: (number = 1)"
"        ->  Bitmap Index Scan on room_hotel_id_fkey  (cost=0.00..960.77 rows=51798 width=0)"
"              Index Cond: (hotel_id = 2)"
```

Reversed-key Index

- Specialty by Oracle
- Increases index updates throughput
  - Number of insertions / updates per second
- Idea
  - Key values are reversed in index
  - $\rightarrow$ sequence-generated values are scattered
    - E.g., 12345 and 12346 $\rightarrow$ 54321 and 64321
  - $\rightarrow$ diminishes collisions in concurrent index updates
- CREATE INDEX idx ON tab(attr) REVERSE;

Global (Schema) Changes

- Creating indexes
- Schema change
  - See next slides
- Relation partitioning
  - See next slides

Lecture Takeaways

- Pure predicates vs functional indexes
  - Time with time zone issues
- Avoid unnecessary statements
- Do not overuse temp tables
- Mind impacts of new indexes