Query Tuning

Outline

- Understanding Query Plans
  - Physical Operators
  - Cost Model
  - Query Optimization
- Query Tuning
  - Rewriting Methods
  - Aggregate Maintenance
  - Tuning Triggers

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?

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

External Sorting

- Sorting is used in implementing many relational operations
- Problem:
  - Relations are typically large, do not fit in main memory
  - So cannot use traditional in-memory sorting algorithms
- Approach used:
  - Combine in-memory sorting with clever techniques aimed at minimizing I/O
  - I/O costs dominate => cost of sorting algorithm is measured in the number of page transfers

External Sorting (cont'd)

- External sorting has two main components:
  - Computation involved in sorting records in buffers in main memory
  - I/O necessary to move records between secondary storage and main memory

Duplicate Elimination

- A major step in computing *projection*, *union*, and *difference* relational operators
- Algorithm:
  - Sort
  - At the last stage of the merge step eliminate duplicates on the fly
  - No additional cost (with respect to sorting) in terms of I/O

Access Method

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

Choosing an Access Path

- *Selectivity* of an access path = number of pages retrieved using that path
- If several access paths support a query, DBMS chooses the one with *lowest* selectivity
- Size of domain of attribute is an indicator of the selectivity of search conditions that involve that attribute

Example: $\sigma_{CrsCode=\text{'CS305'}\ \text{AND}\ Grade=\text{'B'}}$ (Transcript)

- a B^+ tree with search key *CrsCode* has lower selectivity than a B^+ tree with search key *Grade*

Computing Joins

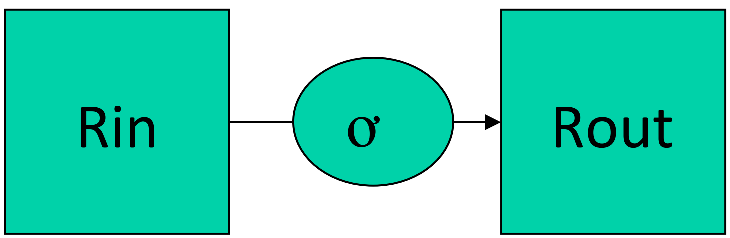
- The cost of joining two relations makes the choice of a join algorithm crucial
- Simple *block-nested loops* join algorithm for computing $r \bowtie_{A=B} s$

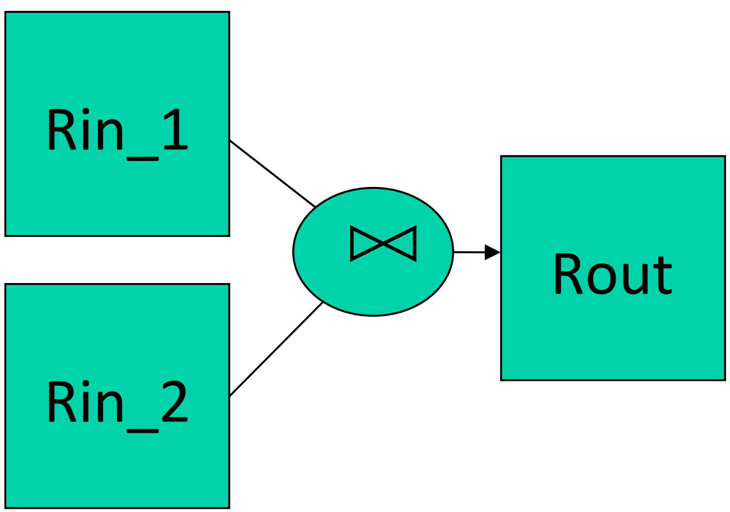
foreach page p_r in r **do**
  foreach page p_s in **s** **do**
    output $p_r \bowtie_{A=B} p_s$

Join Algorithms

- Nested Loop
- Block-Nested Loop
- Index-Nested Loop
- Hash Join
- Sort-Merge Join

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 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)
```

Query Representation

- Query Tree

sname (On-the-fly)

sid=sid (Sort-Merge Join)

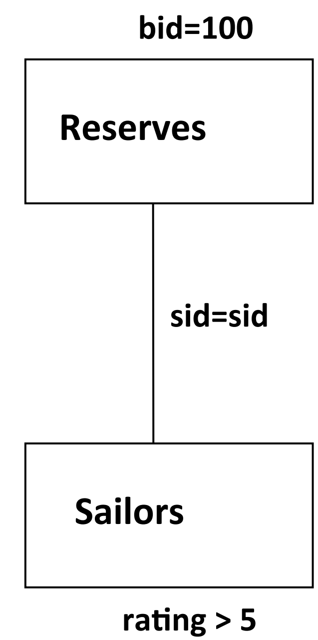
(Scan; write to temp T1) bid=100

rating > 5 (Scan; write to temp T2)

Reserves

Sailors

- 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)

Search Algorithm

<u>Greedy</u>

- Cost model
  - Based on statistics (size of relation, distribution of attribute values)
  - I/O cost of each operator
- Choice of join order using a greedy approach
  - For each outermost table
    - Find best join operator with one of the remaining table
    - Repeat until no remaining table
  - Keep best plan (left deep plan)

Search Algorithm

<u>Dynamic programming (System R)</u>

- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
- For each subset of relations:
  - Cheapest plan overall,
  - Cheapest plan for each *interesting order* of the tuples.

Query Monitoring

- Two ways to identify a slow query:
  - It issues far too many disk accesses, e.g., a point query scans an entire table
  - Its query plan, i.e. the plan chosen by the optimizer to execute the query, fails to use promising indexes

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

Running Example

- Employee(ssnum, 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(ssnum, name, degree_sought, year)
  - Clustering index on ssnum
  - Non clustering index on name
  - Ssnum determines all the other attributes
- Tech(dept, manager, location)
  - Clustering index on dept; dept is key.

Query Rewriting Techniques

- Index usage
- DISTINCTs elimination
- (Correlated) subqueries
- Use of temporaries
- Join conditions
- 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;
    ```

  - 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 is unnecessary, since ssnum is a key of employee so certainly is a key of a subset of employee.

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?

Distinct Unnecessary Here Too

- Since dept is a key of the tech table, each employee record will join with at most one record in tech.
- Because ssnum is a key for employee, distinct is unnecessary.

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: Proof Sketch

- If every relation is privileged then there are no duplicates
  - The keys of those relations are in the from clause
- Suppose some relation T is not privileged but reaches at least one privileged one, say R. Then the qualifications linking T with R ensure that each distinct combination of privileged records is joined with at most one record of T.

Reaches: Example 1

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

- The same employee record may match several tech records (because manager is not a key of tech), so the ssnum of that employee record may appear several times.
- Tech does not reach the privileged relation employee.

Reaches: Example 2

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

- Each repetition of a given ssnum vlaue would be accompanied by a new tech.dept since tech.dept is a key of tech
- Both relations are privileged.

Reaches: Example 3

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

- Student is priviledged
- Employee does not reach student (name is not a key of employee)
- DISTINCT is needed to avoid duplicates.

Types of Nested Queries

- Uncorrelated subqueries with aggregates in the nested query

  SELECT ssnum FROM employee
  WHERE salary > (select avg(salary) from employee)

- Uncorrelated subqueries without aggregate in the nested query

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

- 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)

- Correlated subqueries without aggregates

  (unusual)

Rewriting of Uncorrelated Subqueries without Aggregates

- Combine the arguments of the two FROM clauses
- AND together the where cluases, repacing in by =
- 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
```

Rewriting of Uncorrelated Subqueries 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 of Correlated Subqueries

- Query: find the employees of tech departments who earn exactly the average salary in their department

```
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);
```

Rewriting of Correlated Subqueries

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

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

Rewriting of Correlated Subqueries

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

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

Rewriting of Correlated Subqueries

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

- ```
  SELECT ssnum
  FROM employee, temp
  WHERE numfriends = numcolleagues
      AND employee.dept = temp.dept;
  ```

- Can you spot the infamous COUNT bug?

The Infamous COUNT Bug

- Let us consider Helene who is not in a technical department.
- In the original query, helene's number of friends would be compared to the count of an empty set which is 0. In case helene has no friends she would survive the selection.
- In the transformed query, helene's record would not appear in the temporary table because she does not work for a technical department.
- This is a limitation of the correlated subquery rewriting technique when COUNT is involved.

Abuse of Temporaries

- Query: Find all information 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'
- 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 numerical attributes rather than on string attributes.

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;

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;
- Optimizers expand views when identifying the query blocks to be optimized.

- The selection from techlocation is expanded into a join:
- SELECT location
  FROM employee, tech
  WHERE
  employee.dept = tech.dept
  AND ssnum = 43253265;

Aggregate Maintenance

- The accounting department of a convenience store chain issues queries every twenty minutes to obtain:
  - The total dollar amount on order from a particular vendor
  - The total dollar amount on order by a particular store outlet.

- Original Schema:
  Ordernum(ordernum, itemnum, quantity, purchaser, vendor)
  Item(itemnum, price)

- Ordernum and Item have a clustering index on itemnum

- The total dollar queries are expensive. Can you see why?

Aggregate Maintenance

- Add:
  - VendorOutstanding(vendor, amount), where amount is the dollar value of goods on order to the vendor, with a clustering index on vendor
  - StoreOutstanding(purchaser, amount), where amount is the dollar value of goods on order by the purchaser store, with a clustering index on purchaser.
- Each update to order causes an update to these two redundant tables (triggers can be used to implement this explicitely, materialized views make these updates implicit)
- Trade-off between update overhead and loopup speed-up.

Materialized Views in Oracle9i

- Oracle9i supports materialized views:

```
CREATE MATERIALIZED
VIEW VendorOutstanding
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT orders.vendor,
sum(orders.quantity*item.price)
FROM orders,item
WHERE orders.itemnum =
item.itemnum
group by orders.vendor;
```

- Some Options:
  - BUILD immediate/deferred
  - REFRESH complete/fast
  - ENABLE QUERY REWRITE
- Key characteristics:
  - Transparent aggregate maintenance
  - Transparent expansion performed by the optimizer based on cost.
    - It is the optimizer and not the programmer that performs query rewriting

Triggers

- A trigger is a stored procedure (collection of SQL statements stored within the database server) that executes as a result of an event.
- Events are of two kinds:
  - Timing events
  - Modifications: inserts, deletes or updates
- A trigger executes as part of the transaction containing the enabling event.

Reason to Use Triggers

- A trigger will fire regardless of the application that enables it.
  - This makes triggers valuable for auditing purposes or to reverse suspicious actions, e.g. a salary changed on a Saturday.
- Triggers can maintain integrity constraints, e.g., referential integrity or aggregate maintenance
- Triggers are located on the database server and are thus application independent.

Life without Triggers

- Application must display the latest data inserted into a table
- Without triggers, the application must poll data repeatedly:
  - SELECT *
    FROM table
    WHERE inserttime >= lasttimelooked + 1;
  - Update lasttimelooked based on current time.
- Poll too often and you cause lock conflicts.
- Poll too seldom and you will miss updates.

Triggers Can Help

- Implement an interrupt-driven approach:
  - ```
    CREATE TRIGGER todisplay
    ON table
    FOR insert AS
    SELECT *
    FROM inserted
    ```
- The trigger avoids concurrency conflicts and provide new data exactly when produced.
    ```

Problems with Triggers

- In the presence of triggers, no update can be analyzed in isolation, because triggers might cause further updates.
- The interaction between triggers might be hard to maintain when the number of triggers grow:
  - A modification might potentially fire several triggers: which ones are fired? in which order?

Tuning Triggers

- A triggers executes only after a given operation has taken place. The programmer might make use of this information to suggest a specific execution strategy.
  - Non clustered indexes to speed-up referential integrity if records are inserted one at a time.

Tuning Triggers

- It is easy to write triggers that execute too often or return too much data.

  Example: write records to table Richdepositor everytime account balance increases over $50000.

  - A naïve implementation:

```
CREATE TRIGGER nr
ON account
FOR update
AS
INSERT INTO Richdepositor
FROM inserted
WHERE inserted.balance >
50000;
```

- A more efficient implementation:

```
CREATE TRIGGER nr
ON account
FOR update
AS
if update(balance)
BEGIN
  INSERT INTO RichDepositor
  FROM inserted, deleted
  WHERE inserted.id = deleted.id
  AND inserted.balance > 500000
  AND deleted.balance < 500000
END
```