Solutions to

Final Exam 2012-08-21
DATABASE DESIGN II
1DL400

Date: Tuesday August 21, 2012
Time: Home
Teacher on duty: Tore Risch, phone 471 6342 or 070 425 0522
Exam aid: course material

Instructions:

- Read through the complete exam and note any unclear directives before you start solving the questions. The following guidelines hold:
  - Write clear and short answers! Answers that cannot be read can obviously not result in any points and unclear formulations can be misunderstood.
  - Assumptions outside of what is stated in the question must be explained. Any assumptions made should not alter the given question.
  - Write your answer on only one side of the paper and use a new paper for each new question to simplify the correction process and to avoid possible misunderstandings.
- A passing grade requires about 50% of the maximum number of points.

GOOD LUCK!
1. Indexing
   a. What is a scan? (1p)
      A scan is a data structure to enable iterating over a stream of tuples returned by some query operator without materializing the result.
   b. What are the main scan operators for a B-tree index? (1p)
      OPEN INDEX SCAN FOR GIVEN KEY (to scan all rows in an interval of keys)
      NEXT
      CLOSE
      OPEN FULL SCAN
      In addition there are usually update operators at current scan position:
      UPDATE
      DELETE
   c. What are the main scan operators for a hash index? (1p)
      OPEN INDEX SCAN FOR GIVEN KEY (to scan all rows for a given key)
      OPEN FULL SCAN
      NEXT
      CLOSE
      In addition there are usually update at current scan position:
      UPDATE
      DELETE
   d. How are scans used with JDBC? (1p)
      The result a query submitted with JDBC is internally represented as a scan and encapsulated in the JDBC datatype ‘rowset’. By iterating over a rowset the scan is moved forward. This avoids saving large data structures representing query results in Java’s heap.
   e. How do execution plan operators use scans? Why? (2p)
      All intermediate results in query operators are represented by scans. The intermediate results are not materialized except when using execution plan operators that explicitly materialize the result. The reason is that materialized representation of intermediate results that can be very large should be avoided to save space.

2. Relational Calculus
   We have two relations:
   
   EMPLOYEE(SSN, NAME, SALARY, DEPT, ADDRESS, PHONE)
   DEPARTMENT(DNO, DNAME, DIVISION)

   Translate the following SQL query to tuple relational calculus (2p):

   SELECT SSN, NAME, PHONE, DNAME FROM EMPLOYEE, DEPARTMENT WHERE DIVISION = "Travel" AND DEPT = DNO AND SALARY > 3000
{e.ssn, e.name, e.phone, d.dname | employee(e) ^
department(d) ^ d.division="Travel" ^ e.dept=d.dno ^
e.salary>3000}

3. Query Optimization
   A relational database has the table:

   EMPLOYEE(PNR, SALARY, SEX, DEPT), key PNR

   The cost-based optimizer uses the following statistics stored as meta-data in the
   DBMS's system tables:

   - There are 10000 rows in the table.
   - There are 20 departments in the company.
   - SEX is either 'Male' or 'Female'.
   - The lowest salary is 20000 and the highest is 100000.

   Statistical assumptions:
   - The query optimizer assumes flat distributions of data values in columns
   - The query optimizer assumes independence between values in different columns.

   a. What is the selectivity of the condition
      SALARY < 25000  (1p)
      0.0625: (25000-20000)/(100000-20000)

   b. What is the selectivity of the condition
      SALARY>100000 OR SEX != 'Male'  (1p)
      0.5 since there is no salary > 100000.

   c. What is the selectivity of the condition
      ((SEX= 'Male' AND DEPT = 'Toys') OR
       (SEX= 'Female' AND DEPT = 'Tools')) AND
       (SALARY > 50000 OR SALARY <25000) (2p)
      s1 = 0.5 * 1/20 = 0.025 (SEX= 'Male' AND DEPT = 'Toys')
      s2 = 0.5 * 1/20 = 0.025 (SEX= 'Female' AND DEPT = 'Tools')
      s3 = (100000-50000)/(100000-20000) = 0.625 (SALARY > 50000 )
      s4 = (25000-20000)/(100000-20000) = 0.0625 (SALARY <25000)
      combined: (s1 + s2 - s1*s2)*(s3 + s4 - s3*s4) = 0.032

4. NoSQL databases
   a. What is eventual consistency and how does it differ from regular database
      transactions? (1p)
      Eventual consistency means that the database consistency is not guaranteed
after commit, but after some unspecified time after each commit. If there are many commits the time to reach consistency will be further delayed.

b. What are the **pros and cons** of eventual consistency? Give **examples**. (2p)
The advantage with eventual consistency is performance of the commit operation. In a fully consistent distributed database, to make a full consistent commit requires all (or a majority of) the committing node the exchange synchronization messages (so called 2-phase commit), which can be very expensive. Eventual consistency improves the performance by skipping these exchanges, at the expense of inconsistent databases. If the application requires consistency (e.g. charging, bank accounts, purchases) this can be a very serious limitation!

c. Why is often a simplified SQL used in cloud databases? (1p)
Some cloud databases support only single table access and no joins. Furthermore there are limitations of what kind of search conditions can be made. The reason is that advanced SQL operations are assumed to expensive and they would therefore decrease the overall performance (scalability) of the DBMS. It is assumed that the limited SQL support is sufficient for most applications.

5. Distributed databases
   a. Explain the **differences** between distributed, parallel, and federated databases (2p)
      A distributed DBMS (DDBMS) is a DBMS where different parts of the database is located at different geographic locations (i.e. in several different computers or clusters) but where each part of the database is managed by local instances of the same DDBMS.
      
      A parallel DBMS (PDBMS) is a single DBMS running on a single cluster where the PDBMS engine automatically decides how data is distributed and queries parallelized.
      
      A federated DBMS (FDBMS) is a middleware DBMS that combines data from several different autonomous underlying DBMSs. SQL queries to the FDBMS are more or less automatically translated into queries to the underlying DBMSs.
   
   b. What different kinds of **transparency** are handled by distributed, parallel, and federated databases? (2p)
      In a **distributed database** the location of data is designed manually by the DBA. The distribution is specified using conditions to the DDBMS that declaratively specify how to redistribute data in tables. Thus distributed databases do not have **schema transparency**, since the distributed schema is
designed manually.

In a distributed database queries and updates are specified without need to know the names of the tables in the individual parts of the database. This is called *naming transparency*.

In a distributed database, the user need not know how to split a query or an update into different subqueries to the different parts of the database. This is called *query and update transparency*.

In summary:

*Distributed databases* have query, update, and naming transparency, but no schema transparency.

*Parallel databases* have schema, query, update, and naming transparency. Parallel databases run on clusters, which enables the parallel DMBS to automatically place table fragments.

*Federated databases* have query and naming transparency, but limited updating and no schema transparency. Updating the integrated underlying local databases has to be done locally. In the federated database only tables stored in the global conceptual schema can be updated. All participating database have separate schemas.

c. How is *autonomy* handled by distributed, parallel, and federated databases? (2p)

In *distributed databases* the nodes are all handled by the same DDBMS so there is no autonomy for the participating database parts. The local DBA of a database part has some autonomy the change the local schema though.

In *parallel databases* the PDBMS has full control and there is no autonomy at all for the participating parts.

In *federated databases* each database participating in the federation is its own separate database with its own DBMS and has full autonomy.

6. **DSMS:**

   a. What is the *stride* of a *window* in a continuous query? (1p)

      The stride determines with how many tuples the window jumps forward as the stream progresses. For example, a window holding 10 tuples may have a stride of two tuples.

   b. What is a *tumbling window* in a continuous query? (1p)

      Every time a tumbling window moves its next start point moves to the tuple following the last tuple in the old window. The same stream tuple of the incoming stream will thus never be part of two tumbling windows.
c. What is a **stop condition** and why is it needed in continuous queries but not in regular database queries? (1p)

The stop condition is a predicate indicating that the CQ has ended. It can be, e.g., a time limit, a count, or a user interrupt.

d. Why are **window joins** needed in continuous queries? (1p)

Since data streams are infinite it is impossible to join data streams on any other attribute than the one on which they are ordered (i.e. time stamp or tuple counter). If one wants to join on some other attribute one cannot match (join) attributes in an infinite stream against all values of the same attribute in another infinite stream. Instead window joins are used where the contents of arriving windows in different streams are joined.

e. Why are window join operators **approximate**? (1p)

A correct full window join would require complete scans of the participating streams. Window joins approximate the streams by instead joining sufficiently large windows.
Active databases

The following relation is given:

EMPLOYEE(SSN, NAME, DNO, AGE, PHONE, SALARY)

SSN is social security number,
DNO is department number where employee is working,
SALARY is salary of the employee.

One wants to very quickly retrieve and analyze
some salary statistics for each department and for this one needs
a 'master table':

SALSTAT(DNO, SALSUM, SALAVG, SALMAX, SALMIN)

which for each department maintains the max, min, average and sum of the
salaries.

Show how you can maintain SALDATA by using triggers (ECA-rules) so that the
table SALSTAT is updated correctly whenever the table EMPLOYEE is updated.

Make sure you can handle every possible update of EMPLOYEE that influences
SALSTAT. (5p)

The point with using triggers is to avoid expensive sequential scans of the table
EMPLOYEE to compute the statistics for each update. Thus naive use of triggers
using aggregate functions scanning the entire EMPLOYEE table are not permitted.
We assume that SALSTAT.SALSUM has default value 0.

Situation 1: A new employee is hired:
create trigger salstat_incr after insert on EMPLOYEE
for each row
begin
update salstat set salsum = salsum + new.salary from new where dno=new.dno;
update salstat set salmax = new.salary from new where dno=new.dno and
(new.salary>salmax or salmax is NULL);
/* The is NULL handles the first update */
update salstat set salmin = new.salary from new where dno=new.dno and
(new.salary<salmin or salmin is NULL);
update salstat set salavg = new.salary from new where dno=new.dno; /* 1st
time*/
update salstat set salavg = (salsum + new.salary)/((salsum/salavg)+1) from new
where dno=new.dno; /* since salsum/salavg = N before new employee inserted */
end
Situation 2: An employee resigns:
create trigger salstat_incr after delete on employee
for each row
begin
update salstat set salsum = salsum - old.salary from old where dno=old.dno;
update salstat set salavg = (salsum – old.salary)/(ecnt-1) from old where dno=old.dno;
update salstat set salmax = max(salary) from old where dno=old.dno and old.salary=salmax; /* Notice that this update requires full scan but this happens only where highest paid employee resigns */
update salstat set salmin = min(salary) from old where dno=old.dno and old.salary=salmin; /* Requires full scan only where lowest paid employee resigns */
update salstat set salavg = (salsum-old.salary)/(salavg/salsum-1) from old where old.dno=dno; /*since salavg/salsum is N before employee deleted.*/
end

Situation 3: The salary of an employee is updated is handled by the above two triggers since an update is a delete followed by an insert.

Situation 4: An employee changes department is also handled by the above two triggers since changing department means an deleting the employee from the old department and inserting him to the new.

Situation 5: That a department is removed is also handled by the above two triggers since deleting a department means deleting all employees of the department.