Part C

Basic Part (15 points)

1. [8 points] Consider a relation `Visit(diplomat,country,date)` recording visits diplomats make to certain countries on certain dates. Assume no more than one diplomat visits any given country on a given date. Values for the `date` attribute can be compared using $=$, $<$, $\leq$, etc., and aggregate functions `min`, `max`, and `count` may be applied to the `date` column. Write a SQL query that returns the two most recent visit dates to each country, ordered by date.

   The result of your query should be an ordered two-column relation, where each country appearing in the original `Visit` relation has two adjacent tuples, with the more recent visit for each country listed first. If a country has only one visit, your result should still include a single tuple for that country. For example, we might see the following result, where France has only one visit:

country	date
Afghanistan	4/1/98
Afghanistan	3/31/98
Chile	9/15/97
Chile	8/1/96
France	1/10/98
Saudi Arabia	3/10/97
Saudi Arabia	3/9/97

   If it helps, you may include statements to create views in advance of your one SQL query that computes the final desired result.

2. [4 points] Are the following three schema definitions always equivalent in terms of how the database will behave? If not, how may they differ? Do not consider performance.

```
Schema 1: create table R(A int primary key, B int);
          create table S(C int, D int references R(A));

Schema 2: create table R(A int primary key, B int);
          create table S(C int, D int check(D in (select A from R)));

Schema 3: create table R(A int primary key, B int);
          create table S(C int, D int);
          create assertion A check(
            not exists (select * from S where D not in (select A from R)));
```

3. [3 points] A technique called *rocking* is sometimes used in the implementation of nested-loop or nested-block joins. With rocking, the first time the inner relation is scanned it is scanned forwards, the second time it is scanned backwards, the third time forwards, etc. Under what circumstances is this technique expected to speed up the join and why? Please be brief—the correct answer can be given in 1–2 sentences.

Advanced Part (30 points)

4. [8 points] Consider a relation similar to the one used in Problem #1: `Visit(diplomat, country,start-date,end-date)`, which records visits diplomats make to certain countries starting on `start-date` and ending on `end-date`.

   (a) Write a SQL query to find all diplomats who visited two or more different countries on the same day.

   (b) Now suppose our relation is *temporal*: `Visit(diplomat,country,T)` where `T` is a special *time-interval* attribute that encodes the [`start-date`,`end-date`] interval for each visit. Further suppose we are using a special *temporal relational algebra* in which:

   - Joins automatically intersect time intervals of joining tuples to produce a single time-interval attribute for the joined tuple (and there is no joining tuple if the two time intervals don't intersect).
   - Selection conditions may contain predicates over intervals, e.g., `contains`(*interval1*, *interval2*), `overlaps`(*interval1*, *interval2*), etc.
   - A special constant `NOW` represents the current date.

   Using this temporal relational algebra, write the simplest expression you can think of to find all diplomats who visited two or more different countries on the same day within the last 30 days. Assume that you can refer to the `start-date` and `end-date` components of `Visit.T` values and you can perform arithmetic on days.

5. [4 points] In query languages for *semistructured data*, in order to avoid type errors, predicates of the form "*Expr*$_1$ `comparison-op` *Expr*$_2$" may return `false` (rather than `error`) if the results of *Expr*$_1$ and *Expr*$_2$ are of incomparable types. Briefly explain why a 3-valued logic may be more appropriate to use for such languages if the `not` operator is allowed in predicates. An example is helpful.

6. [6 points] Consider evaluating $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ using hash joins only. Assume as a convention that, when represented as a tree, the *build* operand of a hash join is on the left and the *probe* operand is on the right. Is a left-deep tree or a right-deep tree preferred in order to minimize response time? How about to maximize throughput? Briefly justify your answers.

7. [12 points] Consider three relations $R(A, B)$, $S(B, C)$, and $T(C, D)$ and the following SQL query:

```
select * from R, S, T
where R.B = S.B and S.C = T.C and R.A < 5 and T.D > 10
```

   (a) Draw a relational algebra execution tree for this query, adhering to the following three commonly-used heuristics: (1) *push selections down*; (2) *favor joins over cross-products*; (3) *use a left-deep tree*.

   (b) Give an example scenario where it is more efficient *not* to push one of the selection conditions down for this query. Show the "better" (but equivalent) relational algebra tree and list the characteristics of the scenario (see below).

   (c) Give an example scenario where it is more efficient *not* to favor a join over a cross-product for this query. Show the "better" (but equivalent) relational algebra tree and list the characteristics of the scenario (see below).

(d) Give an example scenario where it is more efficient *not* to use a left-deep tree for this query. Show the "better" (but equivalent) relational algebra tree and list the characteristics of the scenario (see below).

For parts (b)–(d) you should assume that only nested-loop joins are used but they exploit indexes when available. A "scenario" need not be overly detailed or complicated—it can include things like available indexes, relation cardinalities, attribute value distributions, etc.