UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT

DATABASE QUALIFYING EXAM
FALL 1997

Answer four of the following five questions.

1 SQL

Consider the following relations:

Student(*snum:* `integer`, *sname:* `string`, *major:* `string`,
level: `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid*: `integer`, *fname:* `string`, *deptid:* `integer`)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

1. Write the SQL statements required to create the above relations, including appropriate versions of all primary and foreign key integrity constraints.

2. Express each of the following integrity constraints in SQL unless it is implied by the primary and foreign key constraint; if so, explain how it is implied. For each constraint, state what operations (inserts, deletes, and updates on specific relations) must be monitored to enforce the constraint.

   (a) Every class has a minimum enrollment of 5 students and a maximum enrollment of 30 students.
   (b) Every faculty member must teach a course.
   (c) Every course must be taught by a faculty member.
   (d) The total enrollment in courses taught by faculty in the department with *deptid=33* is greater than the number of Math majors.

2 QUERY EVALUATION

Briefly answer the following questions:

1. If the join condition is not equality, can you use sort-merge join? Can you use hash join? Can you use index nested loops join? Can you use block nested loops join?

2. Suppose that you are building a DBMS and want to add a new aggregate operator called `SECOND LARGEST`, which is a variation of the `MAX` operator. Briefly describe how you would implement it.

3. Give an example of how buffer replacement policies can affect the performance of a join algorithm.

3 CONCURRENCY CONTROL

Object-oriented and object-relational database systems provide support for type hierarchies and type extents. Consider a simple type hierarchy in which **Student** is a subtype of **Person** and **Emp** is another subtype of **Person**. Assume that the system maintains automatic type extents, which we will refer to using the same name as the type.

1. In processing a query such as `SELECT * FROM` Person, should hierarchical locking (as defined by Gray) be used? If so, state why; if not, state why not.
2. In addition to the standard set of "base" types (`INTEGER`, `FLOAT`, `STRING`), object-relational systems generally provide reference attributes. For example, the **Person** type might be defined with an attribute *Spouse* whose type is `REF` **Person**. What locking complications does the introduction of reference types introduce? Consider your answer in the context of how the database system implements type extents.
3. Are Gray's solutions to locking hierarchies involving DAGs relevant in the context of reference types?

4 RECOVERY

Briefly answer the following questions:

1. How is checkpointing done in ARIES?
2. Checkpointing can also be done as follows: Quiesce the system so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and transaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointing approach of ARIES?
3. What happens if a second begin_checkpoint record is encountered during the Analysis phase?
4. Can a second end_checkpoint record be encountered during the Analysis phase?
5. Why is the use of CLRs important for the use of undo actions that are not the physical inverse of the original update?
6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

5 SUPPORTING SET-VALUED ATTRIBUTES

Originally the SQL-3 standard included set-valued attributes (most recently this appears to have been postponed to SQL-4!) This simply means that an attribute of a table can be a set of records of some type. The goal of this question is to get your help in deciding how to implement this feature.

For concreteness, suppose that we have the following table declarations:

```
CREATE TABLE Courses ( courseId  INTEGER,
                       preReqs   SET OF INTEGER )

CREATE TABLE Students ( studentId     INTEGER,
                        coursesTaken  SET OF INTEGER )
```

preReqs is the set of ids of prerequisite courses for the given course, and *coursesTaken* is the set of ids of courses taken by the student. A question that must be addressed immediately is whether the elements in a set should be stored internally (along with the rest of the tuple, just like any other variable-length attribute) or externally (in a separate table.) For example, to store the Courses table with the preReqs set stored externally, the system might declare the "hidden" table

```
CREATE TABLE CoursesPrereqs ( courseTupleId    TUPLEID,
                              preReqCourseId   INTEGER )
```

In your answers below you should consider both alternatives.

1. **Simple Indexing:** Suppose you want to create an index to speed up lookups of the sort "Find all students who have taken CS764." That is, a lookup by single set element. What index structure would you propose? How would building and using this index differ from the standard case of building and using a B+tree on an integer attribute?

2. **More Complex Indexing** Suppose now you want to answer queries like "Find all students who have taken the courses CS564 and CS764." That is, lookups that request all tuples whose set contains a given (sub)set. Again, what index structure would you propose? How would building and using this index differ from the standard case of building and using a B+tree on an integer attribute?

3. **Join Algorithms** Suppose you now want to ask the query "Find all (studentId, courseId) pairs such that the student with ID 'studentId' has taken all the prerequisites for the course with ID 'courseId'." That is, you wish to evaluate joins where the join condition is subset containment. Describe algorithms to evaluate this join for the case that (a) both tables have their sets stored internally, and (b) both tables have their sets stored externally. How would you expect your algorithms to perform compared to an equijoin on integer attributes?