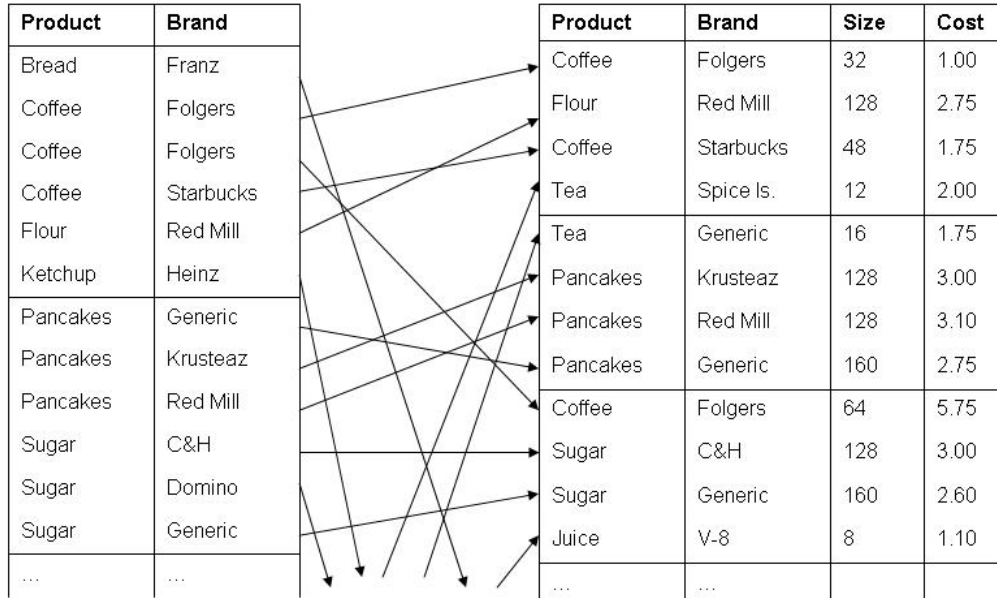


CS 386: Midterm 2, 12 June 2006 Name: _____

You have 1 hour and 50 minutes to do this exam. You may use the course text, materials from the course and a foreign language dictionary, but nothing else. Write your answers on the exam pages – ask the instructor for extra pages if you need them. Work individually. 100 points possible.

Question 1 (12 points): Below are excerpts from a table Inventory and an index defined over that table (index on left). The horizontal lines show the page breaks in both the index and the table.



- a. Is this a clustered or an unclustered index? Explain your answer briefly.

- b. Is this a sparse or dense index? Explain your answer briefly.

- c. Suppose that it takes 600 I/O's to read the entire Inventory table. How many I/O's does it take to read the entire index (just read it, not follow any pointers)?

- d. Which of the following SQL queries is likely to benefit more from this index? Explain why.
 Q1: SELECT * FROM Inventory WHERE Product = 'Tea';
 Q2: SELECT * FROM Inventory WHERE Brand = 'Stash';

Question 2 (10 points): Consider the `Accounts` table below. Suppose it has 1 million rows – too many for a person to read. We want to know whether or not this table currently satisfies the functional dependency `zip → areacode`. Explain how to use one or more SQL queries to test if this FD is satisfied.

`Accounts(name, areacode, phone, zip)`

Question 3 (1 point): Why are there cows on the cover of the textbook?

Question 5 (12 points):

Consider the following sequence of (attempted) SQL commands, where Walt is the creator of the `Sailors` table. For each `SELECT` statement, say whether or not the DBMS will permit it.

Walt: `GRANT SELECT ON Sailors TO Sonia WITH GRANT OPTION`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Sonia: `GRANT SELECT ON Sailors TO Shanti`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Walt: `REVOKE SELECT ON Sailors FROM Sonia RESTRICT`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Walt: `GRANT SELECT ON Sailors TO Shanti`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Walt: `REVOKE SELECT ON Sailors FROM Sonia RESTRICT`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Walt: `REVOKE SELECT ON Sailors FROM Shanti RESTRICT`

Sonia: `SELECT COUNT(*) FROM Sailors`

Shanti: `SELECT COUNT(*) FROM Sailors`

Question 6 (15 points): This question concerns the following relation scheme about breakfast cereals (same as in HW 7).

cereals(CE SE FA PR TC FI SU OC CA FC)

and the FDs

- CE \rightarrow SE FA PR TC FI SU OC CA FC
- FA PR TC \rightarrow CA
- FI SU OC \rightarrow TC
- FA \rightarrow FC

(Abbreviations: CE = cereal, SE = serving size, FA = fat grams, PR = protein grams, TC = total carb grams, FI = fiber grams, SU = sugar grams, OC = other carb grams, CA = Calories, FC = fat Calories)

Decompose this relation into a database schema in BCNF. Try to produce a scheme that preserves the FDs. ***Be sure to indicate clearly which tables are in your final schema and what the key is for each table.***

Question 7 (20 points): Consider the problem of implementing an aggregation operator, with grouping, that can be used to process a query such as

```
SELECT SUM(B) FROM MyRel GROUP BY A
```

One implementation, call it *SortAgg*, first sorts the table on the grouping column(s), then makes a pass through the sorted table and computes the aggregate for each group. (Note that each group is consecutive in the sorted table.) For example, with the query above, we sort MyRel on A, then add up the B-values for each group of rows with the same A-values.

A second implementation, call it *HashAgg*, builds a hash table where the hash key is the value of the grouping column(s). For each group, there will be an entry holding a partial aggregate for that group. Each row is hashed on the grouping value to find its group in the hash table, and then the partial aggregate for that group is updated. Once all the rows are processed, we scan the hash table and output the final aggregate value for each group. For example, with the query above, we hash each row of MyRel on its A-value, then add its B-value to the partial aggregate for its group.

a. Which implementation, SortAgg or HashAgg, do you think runs faster, in general? Why?

b. Which implementation could benefit from a clustered index on the grouping column(s)? How?