CSC 173 Project 4: Database

Ben Kuehnert and Josh Sobel

Build Instructions: Use the included “build” script in order to build the project. This will compile a binary called “database”. Run this binary to see the project run to specification.

What is Displayed: First, the relations as described in FOCS 8.1 and 8.2 are printed out (after they are created). Then, the program outputs informatively each part of the specification in order. Note that Part 2 is implemented with a kind of REPL loop, so in order to see the output of Part 3, you must quit out of both REPL loops (by typing quit 2 times). As a note on notation, queries are given back in the form of sets which are enclosed by curly braces, while tuples are enclosed by parenthesis. In particular, Part 2 will return a singleton set containing a singleton tuple. This is due to the fact that the database code is completely general. So please disregard the extra delimiters on the output. Additionally, if there is no such tuple that satisfies the query, the null set is denoted as {} (open and close curly braces with nothing inside).

Saving and Loading: An example of saving/loading is in main.c lines 105-127. The program creates a relation, saves it to file, then immediately loads it and prints it. Feel free to (after running the program to generate the saved file) comment out the parts that create the relation (lines 106-123), and you will see that the relation saved to file is loaded.

Implementation of Basics: The basic relation keeps track of tuple size, the primary index/hashtable, and an array of secondary indexes with their hashtables. Insert simply takes the inputted tuple, and gives it to each hashtable (our hashtable knows how to hash tuples based on a given vector, which defines what index it keeps track of). Lookup does a small computation to figure out if we have an index on the tuple we are querying, if so it uses it. If not, it completes a brute force search. Our relation doesn’t actually allow completely general indexes (each index must only hash one value), as then different indexes would be members of the powerset of the attributes, which would be difficult to use effectively. So, as a slight departure from the generalization: each index keeps track of exactly one attribute, and queries can use multiple indexes and use set intersection to find the answer, which is much faster than brute force. Deletion is done similarly, however we need to go through each hashtable and delete, which for each hashtable is essentially a lookup (with restraints on access to the other hastables).

Implementation of Relational Algebra: The selection operator uses the fact that a selection call with an arbitrary boolean condition can be expressed with composition of selection and the union operation, where selection condition can only be of the form “A = a” or “A != a”. The implementation is simple. If the relation has an index on A, use it to quickly compute the set of tuples that satisfy the condition. If not, brute force. The projection operator iterates through each tuple of the relation and then adds the tuple with only the projected attributes. Since the relation avoids adding duplicates this new relation is the projection and is returned. The join operator iterates over all tuples in one relation and compares it against each tuple in the other relation. When the components are same in the columns being joined the tuple is added to a new relation. This new relation is returned as the join of the two relations.

Extra Credit: This code is extremely general. It can handle any schema, and effectively any index design. It makes absolutely 0 assumptions on the input data (except that it will be in the form of a char pointer, but that is hardly a restriction), and was able to handle the example data without any modification at all.

Relational algebraic operations are much the same, and just consider the relation as an abstract data type.