SUR: A SINGLE USER RELATIONAL DBMS [1]

ABSTRACT

This report describes SUR, a single user relational database management system. SUR is designed to offer relational database facilities to a host language – JAVA, C++, LISP, RUBY, PYTHON on a personal computer. SURLY, the SUR language facility, consists of a data definition language (DDL) and a relationally complete data manipulation language (DML). SURLY is designed to be extensible so that after a core of basic commands and utility routines have been implemented, new commands can be added to SURLY in a modular way.

[1] If you are reading this document online in MS Word, then turn on View/Document Map to see and move around on the outline.

TABLE OF CONTENTS

LIST OF FIGURES

I. Core of SUR AND SURLY

A. SUR – Introduction

Data is recognized to be an important long-term asset of an enterprise (company, individual, ...). Where formerly the emphasis was on libraries of programs that manipulate files of data according to each program's particular view of the data, now increasingly enterprises are recognize the importance of representing large amounts of data in a uniform way in a central, formatted database. Advantages of this arrangement – availability of data, redundancy control, are well documented in textbooks on database management systems by Elmasri and Navathe, Date and many others.

A data model is a representation for data which is based on a small number of data object primitives and a well defined collection of primitive operations on the data objects. Over the years, several important data models have emerged in the database area – the hierarchical model is based on trees, the network model is based on graphs, the relational model is based on tables/ mathematical relations, the object database model is based on OOP; other data models include or are based on entity-relationship, unified modeling language (UML), XML, and grid technology. Of these, the relational data model has been the industry workhorse over the last 30 years because it is simple and easy to work with. Relational query languages are easy to learn and complex queries are expressible in a straightforward way in terms of powerful operators on tables.

This report describes a small but powerful relational database management system called SUR. SUR, a single user relational DBMS, is designed to offer relational database facilities to a host language – C++, Java, LISP, … on a personal computer. SUR is useful in DBMS environments where (single) relations are never too large to fit in a program's address space. This, of course, is a major restriction for large applications, but SUR has clear applicability for middle and small scale applications where large relations have fewer than O(many thousand) tuples. These applications include individual and small enterprise databases for small businesses, hobbyists, and home computers, The SUR language facility, SURLY, consists of a data definition language (DDL) and a data manipulation language (DML). The SURLY DDL has facilities for creating and destroying relations and secondary indices. The SURLY DML allows easy insertion and deletion of tuples and printing of relations, and offers a relationally complete algebra including relational assignment as well as the nestable operators UNION, SET-DIFFERENCE, PROJECT, SELECT, and JOIN. In a modular way, SURLY can be extended to include VIEWS, TRIGGERS, simple INTEGRITY CONSTRAINTS, LINKS, READ/WRITE, the calculus statement RETRIEVE, and a hierarchical formatted PRINT statement.[2] SURLY can be used as a batch or interactive language. A near variant ESURLY can be used executably. At the implementation level of SUR, memory is divided into NAME SPACE and POINTER SPACE

[2] Relational algebra and calculus are due to Codd. The syntax of SURLY's algebra was suggested by Date. The RETRIEVE command is similar to the INGRES-QUEL RETRIEVE. The ideas for VIEWs, TRIGGERs, INTEGRITY CONSTRAINTs and LINKs are taken from System R.

(similar to LISP's FULL- and FREE-WORD space). NAME SPACE stores components of relations uniquely as strings. POINTER SPACE is further divided into a list space for tuples, relation heaps, and trees, and a linear address space for hash tables.

This report is arranged in four sections.

- Section I covers SURLY's RELATION, DESTROY, INSERT, DELETE, INPUT, and PRINT statements. Also the HEAP index (a linked list).
- Section II adds INDEX for storage structures TREE and HASH to make querying more efficient, as well as EXPORT and IMPORT statements.
- Section III covers the relational algebra: PROJECT, SELECT, JOIN, UNION, INTERSECTION as well as DELETE WHERE.
- Section IV is a list of extensions to the basic SURLY language. No computer implementation of SUR is described here. Instead, suggestions for an implementation are provided.

The scale and nature of SURLY gives students an opportunity to put into practice principles learned in this and earlier courses (especially structured programming and data structures).

B. The Syntax of SURLY

The syntax of the SURLY language can be described in a variant of BNF as shown in Figure 1. Figure 2 shows sample data from the CSCI_DEPARTMENT database. Students should feel free to modify the input language to be consistent with the implementation language they choose.[3]

Metasymbols

```
::=     means is defined as
{x}     means repeat x zero or more times
[x]     means x is optional
x | y   means either x or y
x || y  means concatenate x to y
<x>     means x{[,]x} e.g. x x x ...or x,x,x
```

[3] So for instance an INSERT command in LISP might appear as:

```
(INSERT "COURSE" @(CS161O "STRUCTURED PROGRAMING IN PL/I" 3)).
```

Rules

```
surlyinput              ::= {command}
command                 ::= RELATION relationname (<attrib format>); |
                            INDEX indexname ON relationname
                              ORDER attriblist
                              STORAGE STRUCTURE storagestructure; |
                            DESTORY relname; |
                            INSERT relationname tuple; |
                            DELETE relationname [WHERE (qualifier)]; |
                            INPUT {relationname {tuple;} *} END_INPUT; |
                            PRINT <relexpr>; |
                            EXPORT <relationname>; |
                            IMPORT <relationname>; |
                            relationname = relexpr;
relexpr                 ::= relname |
                            JOIN relexprl AND relexpr2
                              OVER attriblist1 AND attriblist2 |
                            PROJECT relexpr OVER attriblist |
                            SELECT relexpr WHERE (qualifier) |
                            UNION relexprl AND relexpr2 |
                            SET_DIFFERENCE relexprl AND relexpr2 |
                            COPY relexpr |
                            ASSIGN relationname = relexpr |
                            PRINT relexpr |
                            (relexpr [GIVING attriblist])
storagestructure        ::= HEAP | HASH | TREE  (HEAP is default)
format                  ::= CHAR length | NUM length
length                  ::= an integer
relname                 ::= relationname | indexname
relationname            ::= identifier
indexname               ::= identifier
attrib                  ::= identifier
attriblist              ::= (<attrib>) | attrib
tuple                   ::= <value>
value                   ::= character string varying
identifier              ::= letter || {letter | number | _}
                               --length is implementation dependent
qualifier               ::= qualifier1 | qualifier1 OR qualifier
qualifier1              ::= compare | compare AND qualifier1
compare                 ::= attrib relop value
relop                   ::= = | < | <= | >= | > | !=
comment                 ::= /* comment */
```

Figure 1. The Syntax of SURLY

SURLY Test Data (Sample Test File)

```
/* SURLY COMMAND FILE CONTAINING THE SAMPLE DATABASE SCRIPT */
RELATION COURSE (CNUM CHAR 8, TITLE CHAR 30, CREDITS NUM 4);
RELATION PREREQ (CNUM CHAR 8, PNUM CHAR 8);
RELATION OFFERING (CNUM CHAR 8, SECTION NUM 5, STARTHOUR CHAR 5, ENDHOUR
CHAR 5, DAYS CHAR 5, ROOM CHAR 10, INSTRUCTOR CHAR 20);
RELATION STAFF (NAME CHAR 20, SPOUSE CHAR 10, RANK CHAR 5, CAMPUSADDR CHAR
10, EXTENSION CHAR 9);
RELATION INTERESTS (NAME CHAR 20, INTEREST CHAR 30);
RELATION DEPT (NAME CHAR 20, DEPT CHAR 4);
INSERT COURSE CSCI141 'COMPUTER PROGRAMMING I' 4;
INSERT COURSE CSCI145 'COMP PROG & LINER DATA STRUCT' 4;
INSERT COURSE CSCI241 'DATA STRUCTURES' 4;
INSERT COURSE CSCI301 'FORMAL LANGUAGES' 5;
INSERT COURSE CSCI305 ALGORITHMS 4;
INSERT COURSE CSCI330 'DATABASE SYSTEMS' 4;
INSERT COURSE CSCI345 'OBJECT ORIENTED DESIGN' 4;
INSERT PREREQ CSCI141 MATH112;
INSERT PREREQ CSCI145 MATH115;
INSERT PREREQ CSCI145 CSCI141;
INSERT PREREQ CSCI241 MATH124;
INSERT PREREQ CSCI241 CSCI145;
INSERT PREREQ CSCI301 CSCI145;
INSERT PREREQ CSCI305 CSCI301;
INSERT PREREQ CSCI305 CSCI241;
INSERT PREREQ CSCI330 CSCI241;
INSERT PREREQ CSCI345 CSCI241;
INSERT OFFERING CSCI141 27921 13:00 13:50 MWF CF115 JAGODZINSKI;
INSERT OFFERING CSCI241 27922 9:00 9:50 MWF AW205 ISLAM;
INSERT OFFERING CSCI241 27935 11:00 11:50 MWF AW403 BOVER;
INSERT OFFERING CSCI305 27950 14:00 14:50 MTWF AW403 LIU;
INSERT OFFERING CSCI330 27974 12:00 12:50 MTWF CF314 DENEKE;
INSERT OFFERING CSCI330 27977 14:00 14:50 MTWF CF316 DENEKE;
INSERT STAFF GREGORY DON SEC A8C 0030;
INSERT STAFF DENEKE WHO ASSIS 'CF 479' 3769;
INSERT INTERESTS DENEKE AI;
INSERT INTERESTS DENEKE DBMS;
INSERT DEPT DENEKE CSCI;
INSERT DEPT ISLAM CSCI;
INSERT DEPT GREGORY MATH;
PRINT COURSE, PREREQ, OFFERING, STAFF, INTERESTS, DEPT;
```

Figure 2. Sample Test Data

C. The Lexical Analyzer

Legal SURLY input consists of SURLY symbols separated by zero or more blanks. SURLY symbols can be defined as follows:

```
'character string containing no single quotes'
character string containing no blanks or break characters
/*comment*/
```

break characters: ( ) = != < <= > >= ; * " ' ,

It is implementation-dependent whether SURLY symbols may be broken across line boundaries. To read your input, write function NEXTSYMBOL which has no arguments, reads and echo-prints the input, skips leading blanks and comments, and returns the next symbol, leaving the "cursor" one position past the end of the symbol (or on the next line}.

For example:

```
NEXTSYMBOL
A: SKIP BLANKS echoprinting;
   CASE current character =
      /*              --find corresponding */; GO TO A;
      ,               --GO TO A; (ignore commas}
      '               --read until matching ' and return string
      ( or ) or = or * --return I ( or ) or = or *
      < or > or ~     --see if = follows and return <= >= ~= or < > ~
      ;               --print carriage return and return ';'
      ELSE            --read until a blank or a break character is
                        encountered and return the string read.
   END CASE;
END NEXTSYMBOL;
```

It might be useful to you to write a function NEXTCHAR(CHAR} which reads and echoprints the next character, setting CHAR to that character and returning that character. NEXTCHAR will then be responsible for keeping track of line boundaries.

D. Basic SURLY Commands

1. RELATION relationname (<attrib format>);

Example.

```
RELATION COURSE (CNUM    CHAR  8,
                 TITLE   CHAR 30,
                 CREDITS NUM   4);
```

Description. RELATION enters a new relation into the database by simply adding a new relation descriptor entry to the CATALOG table. If a relation by that name already exists, DESTROY the old relation before creating the new relation. The new relation is stored as a 'heap' (see "storage structures") and initially contains no tuples. The positional ordering of the attributes in the RELATION definition and the positional ordering of the values in a tuple should correspond one to one (see INSERT}. The format of an attribute consists of the type of the attribute (NUMeric or CHARacter} and the maximum length of the attribute. Character strings longer than "length" characters should be truncated at the right (ERRMSG1}; Numeric strings should contain only digits 0-9 (ERRMSG2) and are truncated to the left if longer than length (ERRMSG3). Both types of data will be stored as character strings, and data values may be shorter than the attributes maximum length.

2. INDEX indexname ON relationname
ORDER attriblist
STORAGE STRUCTURE storagestructure;

Example.

```
INDEX OFFERINGBYINSTRBYCOURSE ON OFFERING
                              ORDER (INSTRUCTOR COURSE)
                              STORAGE STRUCTURE TREE;
```

Description. INDEX is used to create secondary indices on existing relations in order to make retrieval and update with secondary keys more efficient. In order to maintain the integrity of the index, users will not be allowed to update secondary indices directly. However, when a primary relation is changed its secondary indices will automatically be updated by the system. If a DESTROY command is used on the primary relation, all of its secondary indices are destroyed. If a DESTROY command is used on a secondary index, just that index is destroyed. Secondary indices on other indices are not allowed. Secondary indices can be stored as either TREE or HASH storage structures (See "Storage Structures").

3. DESTROY <relname>;

Example.

```
DESTROY COURSE, OFFERINGBYINSTRBYCOURSE;
/*Destroy the COURSE relation and all its secondary indices
  and destroy the OFFERINGBYINSTRBYCOURSE secondary indices*/
```

Description. For each of the relnames specified,

1) if the relname is a secondary index, delete it from the INDEX table and reclaim storage.

2) if the relname is a primary relation, delete it from the CATALOG table, reclaim storage, and DESTROY any secondary indices as in step 1.

4. INSERT relationname tuple;

Example.

```
INSERT COURSE CSCI241 'DATA STRUCTURES' 4;
```

Description. INSERT adds a new tuple to relationname. The relation must already exist and must not be an index. Tuple values must agree in type and order with the corresponding attribute list for the relation, with conversion occurring as specified in the section on the RELATION statement. If relationname has any secondary indices they must be updated as well. If too many or too few tuple variables are encountered in the tuple, an error message is generated (ERRMSG4) and the insertion is aborted.

5. DELETE relationname [WHERE qualifier];

Example.

```
DELETE OFFERING; /*delete all OFFERING tuples*/
DELETE COURSE WHERE CNUM < CSCI2OO AND INSTRUCTOR = DENEKE
   OR CNUM >= CSCI300;
```

Description. DELETE removes tuples which satisfy the qualification (see the discussion on "Qualification") from the relation, reclaims storage, and updates any secondary indices that may exist. If the WHERE clause is not present, the result is to delete all tuples in the relation – the result is a valid but empty relation. Note that DELETE WHERE is related to SELECT WHERE NOT.

NOTE: A relation may be emptied of tuples but not deleted using the DELETE command.

6. INPUT { relationname {tuple; } * } END INPUT;

Example.

```
INPUT COURSE CSCI141 'COMPUTER PROGRAMMING I' 4;
            CMPS304 ALGORITHMS 4;*
INTERESTS DENEKE DBMS;
              " AI;*
END_INPUT;
```

Description. The INPUT command simplifies insertion into relations by:

1) allowing the user to specify sequences of 'relationname tuple tuple... tuple*', without the words INSERT and relationname for each tuple, and

2) allows the user to specify " (one double quote) as a tuple component indicating that the tuple component is the same as the tuple component in the corresponding position of the last tuple encountered. [This ditto feature is optional.]

As in INSERT, too many or too few values in a tuple is an error (ERRMSG4)

7. PRINT <relexpr>;

Example.

```
PRINT COURSE;
```

COURSE		
CNUM	TITLE	CREQ
CSCI141	COMPUTER PROGRAMMING I	4
CSCI241	DATA STRUCTURES	4
CSCI301	FORMAL LANGUAGES	5

Description. PRINT formats and prints in tabular form each of the named relations in its argument list. If a secondary index is specified, PRINT prints the primary relation tuples in the order specified by the secondary index. What action is taken by PRINT when the tuple length exceeds the line length is implementation dependent. Attribute names are truncated to fit the specified format. If a relexpr is not a relname, no relation name is printed. Instead, *TEMPORARY* is printed for the table name.

E. The SURLY Interpreter

The SURLY interpreter has a very simple organization: read an operation, branch to code which reads the operation's arguments, execute the operation and loop back to read the next operation:

```
DO WHILE (TRUE);
   CASE NEXTSYMBOL =
      'RELATION' --create a new relations descriptor in the RELATION
                   class.
      'INDEX'    --create a new index descriptor in the INDEX class; add
                   to the corresponding RELATION.INDICES; and build the
                   new index with the indicated storage structure.
      'INPUT'    --DO WHILE (SETC (RELNAME, NEXTSYMBOL) ~= 'END INPUT');
                     REL# = RELNUM(RELNAME); -
                     DO WHILE (SETN(T, READTUPLE(REL#)) ~= 0);
                        CALL INSERT (REL#,T);
                     END;
                   END;
                   CALL MATCH(';' );
      'INSERT'   --CALL INSERT(SETN(REL#,RELNUM(NEXTSYMBOL)),
                                 READTUPLE(REL#));
                   That is, insert tuple into relation and update any
                   secondary indices.
      'DELETE'    --read RELNAME; if WHERE clause is not present, reclaim
                   storage of all tuples.
      'DESTROY'   --delete all primary tuples and/or all secondary
                   indices, and destroy relation and/or secondary index
                   descriptors
      'PRINT'     --beautifully format the named relations and indices
      ELSE        --print 'BYE BYE' and STOP.
   END CASE;
END DO WHILE;
```

Each of these operations is a module and can be written and tested more or less independently: INPUT calls INSERT, DESTROY calls DELETE, and so you may wish to write a "dummy" INSERT and DELETE to test INPUT and DESTROY. The programming of these modules is fairly straightforward though you will find some hints in the "Programming Notes" section.

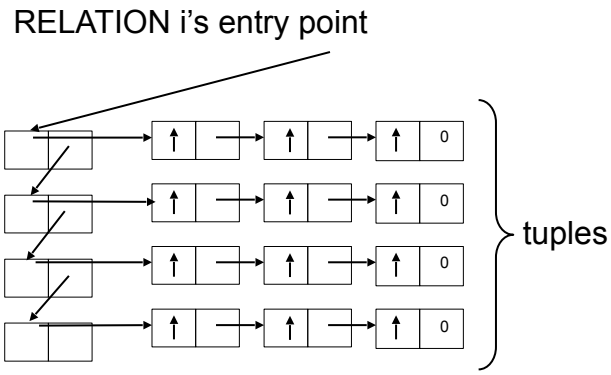
[A high-end alternative is to use a compiler-compiler to read the SURLY grammar specification and translate to code that executes SURLY commands. For instance, see the javacc parser generator.]

G. INDEX statement (storage structures)

Refer to the example of storage structures in Figure 4 (below). The example shows the OFFERING relation stored as a heap (linked list of tuples, where each tuple is also a linked list) and two secondary indices, a bucket hash index called OHASH and a binary tree index called OTREE.

HEAP STORAGE (A LINKED LIST)

Linked list of tuples:



arrows indicate a pointer to a value's unique storage id
that is, a NAME_TBL index

Shown above is a relation with four rows (tuples) and three columns. Remember that our relations do not contain duplicate rows. You can use

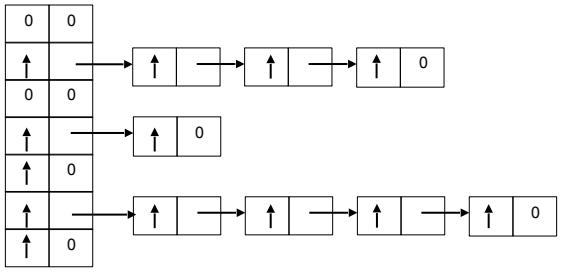
```
MEMBER(TUPLEID,RELATION.ENTRYPOINT(REL#))
```

to determine if a tuple is a duplicate-or-if the relation has an INDEX, you could use the index to determine whether the tuple is already in the relation. This second method is preferable, of course.

HASH STORAGE (BUCKET HASH)

Hash Table for Secondary Index i on Relation j

Linear address space



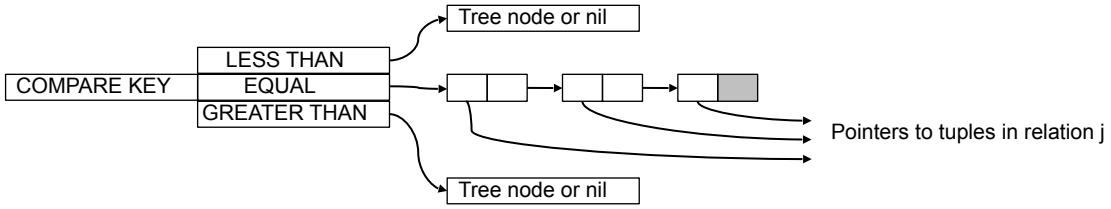
arrows indicate a pointer to a tuple in Relation j

Hash storage requires a linear address space of size N, and so hash tables reside at the upper end of POINTER SPACE. The hash access method serves to group tuples with the same key together (as well as other tuple-key combinations hashing to a giver address). Access to a tuple in hash storage is fast, about order (number of tuples in the relation/hash table size). So the hash table size should be approximately the same as the number of the tuples in the relation. To simplify hash table allocation, make all hash tables the same size --the average size of a relation, say 50. (So access is order (1).) The hash function should provide as uniform a spread as possible; you may wish to base it on the key values themselves or alternatively on the unique storage ids (NAMETBL indices) of those values. One example of a hash function is:

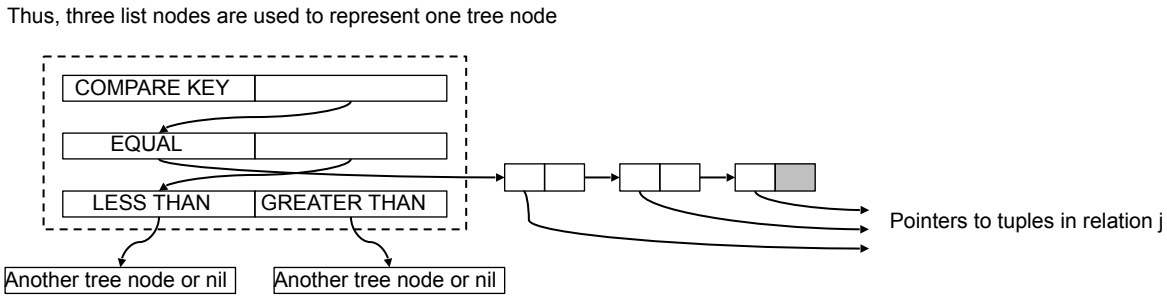
```
HASH_INDEX = REMAINDER((Σ KEYSTORAGEIDS)*LARGEPRIME#, HASHTBLSIZE) + 1;
```

TREE STORAGE (A BINARY TREE)

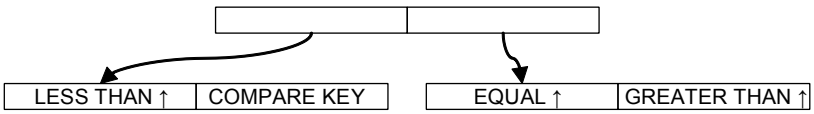
TREE storage will be implemented with a binary tree, so nodes might look like the following:

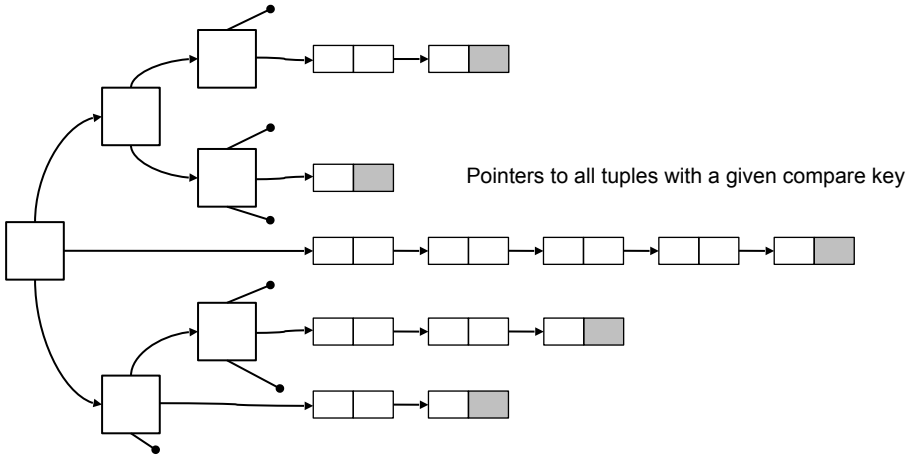


But, for uniformity, a node can be represented with a linked list as [4]:



[4] Alternatively, the node structure below speeds up tree traversals but makes node garbage collective a bit harder:





The main purpose of an TREE index is to order a relation with respect to one or more attributes, here called the ORDER of the index. At present, only ASCENDING orders are implemented in SURLY. The compare key of a tree node can be constructed as a linked list pointing to a tuple's ORDER attributes in NAMESPACE. (“Programming Notes".)

For example, consider the indices below. Figure 4 shows how they might be represented.

```
INDEX OHASH ON OFFERING KEY(INAME)       /* see left  side of Fig 4 */
            STORAGE_STRUCTURE HASH;
INDEX OTREE ON OFFERING KEY (INAME CNUM) /* see right side of Fig 4 */
            STORAGE_STRUCTURE TREE;
```

Now consider:

```
INSERT OFFERING CSCI591 NELSON A3OOO TR9;
DELETE OFFERING;
PRINT  OFFERING;
PRINT  OTREE;
PRINT  OHASH;
DELETE OFFERING;
```

An alternative to the storage organization described here uses a uniform, parameterized hash/ tree index. Hash buckets are binary trees, not linked lists. Hash tables are of size 50 for HASH indices and of size 1 for TREE indices.

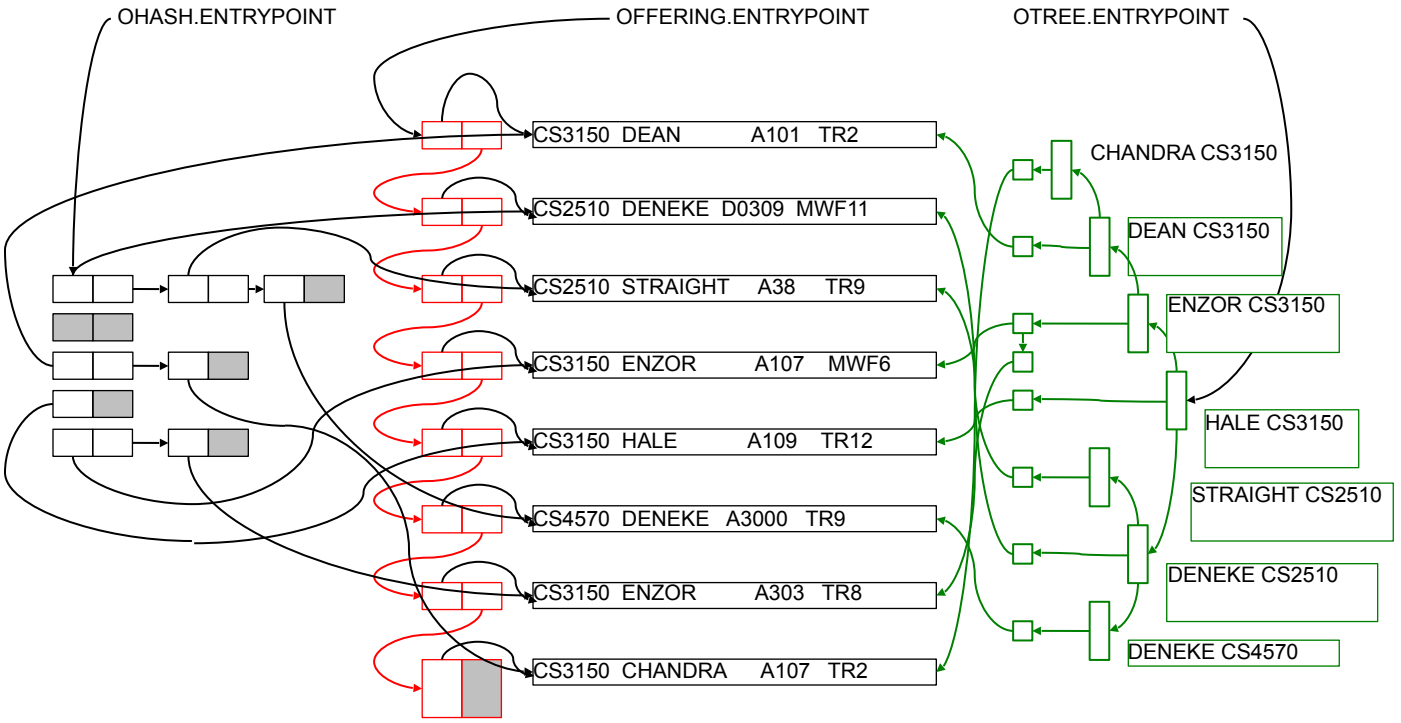


Figure 3: Secondary Indices in SURLY

H. EXPORT and IMPORT statements

EXPORT <relationname>;

For each relation in the list of relations, export the relation to a file of the same name (in a default directory).

To export a relation, make up an XML format that encodes the RELATION, INDEXes, and tuples for a relation and write that to the file.

IMPORT <relationname>;

For each relation in the list of relations, import the relation from a file of the same name (in a default directory). That means reading the XML statement.

(Alternatively, you can export a file by writing RELATION, INDEX, and INPUT statements and importing is just reading these. In.

I. Assignments

This looks like a healthy project – but we have a whole quarter. Still, the best way to attack a big problem is to break it into sub-problems. For this kind of problem, a natural decomposition is to implement only a subset of the system's rules. Then to add on modules as we go until the whole language has been implemented. The most natural set of rules to start with is one that implements a subset of SURLY.

Do not write and test your program all at once. You have several weeks to finish your assignment, but you won't finish it if you wait to the last week to get started. First break the problem into pieces and decide to work on certain pieces each week. Test each piece with a driver before putting the pieces together.

Consider that you are implementing a prototypical SURLY. In this version, don't spend a lot of time on handling input errors. You may assume error-free input.

Be sure to include a comprehensive collection of test cases with your assignment. Develop a file of test commands and read these in, executing them. This “regression test” will help you test your program.

II. Relational Algebra Commands

First we describe the rest of the basic SURLY commands.

A. The Relational Assignment Statement and Relexpr's

```
relationname = relexpr;
```

REL EXPR

Let RELNUM be the index of a relation in the RELATION table. Consider implementing the function RELEXPR of no arguments, which reads relation- expressions composed of JOIN's, PROJECT's, relname's, ...and returns a RELNUM pointing to a new temporary relation. For example:

```
RELEXPR =
CASE NEXTSYMBOL =
   'JOIN' DO;
               RELNUMl = REL EXPR; MATCH ('AND' );
               RELNUM2 = REL-EXPR; MATCH ('OVER')'
               ATTRIBNUM1 = GETATTRIBNUM (RELNUM1, NEXTSYMBOL);
               MATCH('AND')
               ATTRIBNUM2 = GETATTRIBNUM (RELNUM2, NEXTSYMBOL);
               RETURN (JOIN (RELNUM1, RELNUM2,
                             ATTRIBNUM1, ATTRIBNUM2));
          END;
   'PROJECT' DO; ...RETURN(PROJECT(RELNUM, ATTRIBLISTPTR)); END;
   'SELECT' DO; ...RETURN(SELECT(RELNUM, QUALIFIER)); END;
   <a relation name> RETURN(RELNUM(the relation name));
   <an index name> RETURN(RELPTR(INDNUM (the index name)));
END CASE;
```

Note that RELEXPR is RECURSIVE. If your implementation language does not have recursion, you may have to simplify JOIN, PROJECT and SELECT to take 'relname's instead of 'relexpr's. The disadvantage is that your SURLY users will have to use lots of relation assignment statements and must now keep track of what to name temporary relations and when to destroy them. Alternatively, you may limp along with some simulation of recursion. Also note: since the functions PROJECT, SELECT and JOIN take relations and not indices as arguments, the user gets no computational advantage from using indexname's instead of relationnames in the corresponding SURLY commands; indexnames are included for convenience. SURLY will decide when to use the indices. Finally, refer back to Figure l and note that relexpr's may be surrounded by parenthesis. This is for convenience when writing large nested expressions. The user is advised to use an indentation scheme when writing large expressions.

The Relation Assignment Statement

The relation assignment statement has the effect of creating a new relation descriptor (or renaming an old one, see below). Now that RELEXPR has been defined, .it should be a simple matter to add the relational assignment to the SURLY interpreter. Just replace the RELNAME of the temporary relation returned by RELEXPR with the name of the left side of the assignment.

Notice that this has the effect of renaming (not copying) a relation if you say relationnamel = relationname2. What happens if you say relationname = indexname? Can two relations with the same name exist in your implementation?

Note that the relexpr (relexpr [GIVING attriblist]) does two things: it allows parenthesis to surround arbitrary relexpr's. Like indentation, this is useful in writing nested relexprs. Secondly, the GIVING clause allows the attributes of the relexpr to be renamed. This is especially useful in JOIN's and RETRIEVE's if non-join fields in two or more relations are similarly named. Add something like the following code to the CASE statement in the definition of RELEXPR above.

```
'('   DO;
         RELNUM = REL EXPR;
         IF MATCH ('GIVING')
         THEN DO; MATCH('(');
                  RENAMEATTRIBS(RELNUM,READATTRIBS);
                  MATCH (')');
              END;
         MATCH (')');
         RETURN (RELNUM);
      END;
```

B. PROJECT

The interpreter, upon recognizing a 'PROJECT' command, goes on to read the projection's arguments and calls:

```
RELNUMNEW = PROJECT ATTRIBLISTPTR FROM RELNUMOLD;
```

The PROJECT function should do the following:

1. Create a new (temporary) relation, replete with attribute names and formats (as in the RELATION statement).

2. If indexes are implemented, create a new (temporary) secondary index with storage structure TREE or HASH, you decide. (Like the INDEX statement). Both structures are initially empty.

3. Now, for each tuple T in RELATION.ENTRYPOINT (RELNUMOLD) do; INSERT (RELNUMNEW, MAKEPROJECTEDTUPLE (T, ATTRIBLISTPTR))

4. If RELATION.TEMPORARY? (RELNUMOLD) = true then DESTROY(RELNUMOLD) and put RELNUMOLD on the RELATION available space list.

The secondary index takes care of

1. Making sure that the new relation has no duplicates.

2. Making insertion relatively inexpensive.

The most obvious MAKEPROJECTEDTUPLE is order (M*N/2) where N = the number of attributes in RELATION (RELNUMOLD) and M = the number of attributes in the Key list. You can do better.

Example.

```
T1 = PROJECT CREDITS, CNUM FROM COURSE;
```

C. JOIN

The function

```
RELNUMNEW = JOIN RELNUM1, RELNUM2 ON ATTRIBLIST_1 <relop> ATTRIBLIST_2;
```

should do the following:

1. Check the join attributes for compatible formats.

2. Create a new (temporary) relation where attribute names are <attribute names from the first relation> then <attribute names from the second relation, minus the repeated attribute name>

3. If indexes are implemented, create a (temporary) secondary index for the new relations.

4. JOIN the two input relations by some efficient algorithm.

- `<relop>` includes: `= | < | <= | >= | > | !=`
- Hint: if they don't already exist, create secondary indices on one or both of the input relations where the ORDER is the JOIN attriblist. How does this help?

5. For each input relation i= 1 to 2;
   If RELATION.TEMPORARY?(RELNUM1) = true
   then DESTROY (RELNUMi) and place RELATION(RELNUMi) on the RELATION available space list.

Example:

if R1(A, B, C) = {<1, 2, 3>, <4, 5, 6>} and R2(D, E) = {<1, 7>, <1, 8>,<4, 9>}, then

JOIN R1, R2 ON A = D would result in

TEMP27(A, B, C, E} = {<1, 2, 3, 7>, <1, 2, 3, 8>, <4, 5, 6, 9>}

Example.

```
J = JOIN COURSE, PREREQ ON CNUM = PNUM;
```

D. SELECT and Qualifiers

The SELECT operation takes a relation and a qualification as arguments and returns a new (temporary) relation as output, composed of all tuples in the original that pass the qualification test. The Qualification is a logical expression composed of AND-ed and OR-ed comparisons where the comparisons are all of the form "attribute relational_operator value". As usual, AND has higher precedence than OR.

You may wish to represent the qualification as an array or struct or class as follows:

```
                        QUALIFICATION ARRAY

    AGE    <  40       OR  AGE   <   40       {AND group 1
AND CNUM   <  CSCI300  AND CNUM  <   CSCI300  {AND group 1
```

```
 OR AGE    >= 40       OR   AGE   >=  40       {AND group 2
AND CNUM   >= CSCI300  AND  CNUM  >=  CSCI300  {AND group 2
AND CNUM   <  CSCI500  AND  CNUM  <   CSCI500  {AND group 2
 OR AGE    =  60       OR   AGE   =   60       {AND group 3
 OR CNUM   != CSCI241  OR   CNUM  !=  CSCI241  {AND group 4
                       STOP
                       …
```

You may wish to write a function TESTTUPLE (TUPLEPTR, QUALIFICATION) which returns "true" or "false" depending on whether the tuple passes the qualification test.

There are a lot of ways to implement the actual SELECTion, ranging from symbolic rearrangement of the qualifier to take optimum advantage of indices, to brute force testing of each tuple in the relation. You are free to implement the selection algorithm any way you wish, but here are some thoughts:

1. Since AND has higher precedence than OR, we can consider a qualifier to be composed of one or more AND-groups which are OR-ed together.
2. If any AND-group contains only compares with "!=" relational operators, then do a linear scan, accepting tuples that pass the AND-group portion of the test and applying the whole test to any tuples which do not.
3. For each AND-group do:
   A. If the AND group contains any compare with an "= " Relational operator and the compare attribute has either an TREE or HASH storage structure, then use the index to find matching tuples and apply the whole qualification test to them.
   B. Set LOWER BOUND and UPPER BOUND = "undefined". If the AND group contains a compare with ">" or ">=" and an TREE index on the compare attribute, set LOWER BOUND to the compare value. If the AND group contains a compare-with a "<" or "<=" on the same attribute, set UPPER BOUND to the compare value. If LOWER BOUND and UPPER BOUND are both defined and LOWER BOUND > UPPER BOUND, go to next AND group. (E.g., IAGE<50 AND AGE>701 can't be true of any tuple.) Otherwise, apply the whole qualification test to all the tuples between LOWER BOUND and UPPER BOUND. (If either is undefined, start at the beginning --or--go to the end.)
   C. Give up examining successive AND-groups and do a linear scan. Since no helpful index exists for the current AND-group, we need to look at every tuple anyway.

If at any point in step 3, the number of tuples inserted into the new relation is greater than say 50% of the number of tuples in the input relation --or --if the number of tuples in the input relation is less than say 25 to start with, then give up and do a linear scan, testing every tuple. (This implies that you associate a cardinality counter with each relation in the CATALOG table, and update it inside of INSERT and DELETE.)

Like PROJECT and JOIN, SELECT first creates a new temporary relation in the CATALOG table, and a new index (if indexes are implemented), then the SELECT logic inserts the selected tuples into the new relation, and finally the input relation is destroyed if it was a temporary.

Example.

```
S1 = SELECT PREREQ WHERE CNUM = CSCI241 OR CNUM > CSCI200
      AND PNUM != MATH200;
```

E. DELETE WHERE

For simplicity in Part I, we ignored the "WHERE (qualifier)" clause of the DELETE command. DELETE with the WHERE clause is obviously similar to SELECT in that qualifying tuples are first selected, then deleted. So, add the WHERE clause to DELETE.

One way to implement DELETE WHERE is to mark tuples but not actually delete them. This has the benefit that secondary indices need not be updated on DELETE. Also, much of the SELECT WHERE logic can be used directly. Two drawbacks of this scheme are:

1) storage for deleted list nodes in the relations heap cannot be reclaimed.

2) access algorithms require extra logic to check whether a tuple has been deleted.

A simple way to mark heap nodes is to set the TUPLEPTR field to 0. Storage for the tuple itself can be garbage collected.

Example.

```
DELETE OFFERING WHERE STARTHOUR != 8:55 AND INSTRUCTOR = BURRIS;
```

F. Intermediate Results – PRINT and ASSIGN

```
relexpr        ::= PRINT relexpr
relexpr        ::= ASSIGN relationname = relexpr
```

Example:

```
Tl = PRINT SELECT OFFERING WHERE (CNUM < CSCI400);
T2= <same example with 'ASSIGN MYTEMP=' replacing 'PRINT'>;
```

Description. Both PRINT and ASSIGN are identity functions in that each returns its evaluated argument relexpr unchanged. PRINT has the side-effect of printing the relexpr. This allows the user to check intermediate results. In the above example, the PRINT allows the user to verify that he is getting all the INSTRUCTOR's that he requested with the SELECT.

ASSIGN has the side-effect of saving intermediate results in a user named relation. The new relation name can be used immediately, even within the relexpr being evaluated. This allows the user to compute common sub-expressions only once. Notice that allowing ASSIGN implies that the user can assume that relexpr's are processed in a particular order (left to right).

This may be an undesirable assumption if SURLY is implemented in an environment that allows process spawning/parallel processing. In some implementations of SUR, the keyword ASSIGN is optional, allowing the compound assignment statement:

```
Rl = R2 = ...= Rn = relexpr;
```

But since SURLY's assignment involves re-naming a relation's descriptor instead of creating multiple descriptors, this statement first creates the temporary relation resulting from evaluating the relexpr, then successively renames it RN ...then R2 then Rl (at which point RN thru R2 are undefined). What are the repercussions in SURLY of using a descriptor-copying instead of a descriptor-renaming semantics for relational assignment?

Clearly both the PRINT and ASSIGN relexpr's are redundant to SURLY's PRINT command and assignment statement. But both commands are useful and require only a little code to implement.

G. UNION, INTERSECTION, SET-DI.FFERENCE, and COPY --optional

```
relexpr ::= UNION relexprl AND relexpr2
relexpr ::= INTERSECTION relexprl AND relexpr2
relexpr ::= SET DIFFERENCE relexprl AND relexpr2
relexpr ::= COPY relexpr
Example: Tl=SET_DIFFERENCE OFFERING AND
(T2=SELECT OFFERING WHERE (CNUM < CSCI500));
T3=UNION T2 AND
         SELECT OFFERING
          WHERE (CNUM >= CSCI500); /* EQUAL?(T1,T3) is TRUE */
T4=INTERSECT Tl AND T2;
T5=COPY Tl;
```

Description: UNION and SET_DIFFERENCE operate on union compatible relations (where corresponding components of tuples agree in type (CHAR/NUM) and degree (-arity)) to return a new relation. Compare this to INSERT and DELETE which modify their relational arguments. COPY returns a COPY of its relational argument.

H. Predicates EQUAL? arid SUBSET? --optional

```
relexpr :: EQUAL?(relexprl, relexpr2}
relexpr ::= SUBSET?(relexprl, relexpr2}
```

Descritpion. Given

```
RELATION TRUE (TRUTH VAL CHAR 9};
RELATION FALSE(TRUTH-VAL CHAR 9};
INPUT TRUE T; * FALSE F; * END_INPUT;
```

EQUAL? and SUBSET? take two union compatible arguments and return a temporary relation EQUAL? to (a copy of} TRUE and FALSE. For example,

```
Tl = SUBSET? (OFFERING, SELECT OFFERING WHERE (COURSE < CSCI400}};
PRINT Tl, EQUAL?(Tl, OFFERING};
```

prints:

T1
TRUTH_VAL
T

TEMPORARY
TRUTH_VAL

F

I. Pseudo-Relation EMPTY – optional

```
relexpr ::= EMPTY
```

Description. Pseudo relation EMPTY is union-compatible with any relation by definition. It is mainly useful in comparing a relation to the empty relation in loops in ESURLY (see below} -e.g. EQUAL?(relexpr,EMPTY} is true if relexpr has cardinality 0. The semantics of EMPTY are:

- if are argument of EQUAL? is EMPTY, EQUAL? returns true iff the other argument is EMPTY or a relation with cardinality 0.
- SUBSET?(relexpr, EMPTY} = True
- SUBSET? (EMPTY, relexpr} = EQUAL?(relexpr, EMPTY}
- if exactly one argument of UNION or if the second argument of SET_DIFFERENCE is EMPTY then return a COPY of the other relexpr
- if both arguments of UNION -or- the first argument of SET DIFFERENCE -or- any argument of JOIN, PROJECT, SELECT, COPY, or PRINT is EMPTY, then return EMPTY
- 'PRINT EMPTY' prints EMFTY as the relation name but no table is printed
- 'ASSIGN relname = EMPTY' is an error (so INSERT, DELETE, ...are undefined on the relation EMPTY}.
- CARDINALITY(RELNUM('EMPTY' }} = 0

J. Executable SURLY (ESURLY)

So far, SURLY is an interpreted language to be read from some file. With a little care, you can write ESURLY, or executable SURLY, which allows a SUR user to write programs containing SUR commands. Consider the following program segment:

```
CALL ASSIGN ('OLD EMPLOYEES',
             PROJECT (SELECT ('EMPLOYEES', 'AGE>30 AND AGE<70')
                     'NAME ADDRESS PHONE AGE'));
CALL PRINT ('OLD_EMPLOYEES');
```

Here we are printing the NAME, ADDRESS, PHONE, and AGE of all employees between 30 and 70 years old. Here, the ESURLY commands, JOIN, SELECT and PROJECT are functions which return relation names of temporary relations. The other SURLY commands are subroutines. All arguments are character strings.

The advantage of ESURLY is that now control structures in the host language can be used to control execution of ESURLY statements. Consider the problem of finding all prerequisites of some course, say CS45l0 (transitive closure). This is a sticky problem in SURLY since you don't know in advance how many JOIN's will be involved. But in ESURLY you might write:

```
CALL RELATION ('RESULT' , 'CNUM CHAR 8');
CALL ASSIGN ('TEMP',
             PROJECT (SELECT('PREREQ', 'CNUM1 = CSCI301'), CNUM2));
DO WHILE (CARDINALITY ('TEMP' ) > 0);
   CALL UNION ('RESULT', 'TEMP');
   CALL ASSIGN ('TEMP' ,
                PROJECT (GIVING(JOIN ('TEMP','PREREQ','CNUM2' ,'CNUM1' ),
                                'CNUM1 CNUM2')
                         CNUM2));
END;
CALL PRINT ('RESULT');
```

Here UNION takes two (union compatible) relation names as arguments, and copies (using INSERT) all tuples in the second relation into the first. CARDINALITY is a function that takes a relationname as its only argument and returns the number of tuples (the cardinality) currently in the relation.

K. Interactive SURLY

Depending on your host language and its host machine, you may wish to implement interactive SURLY, where the input/output files are both the terminal. Unless your interpreter has good error recovery, be careful typing.

You may wish to make some character (say '$') the break character such that if NEXTSYMBOL reads that character it returns to the start of the interpreter loop. Also you will need to define a log out command (say STOP) to gracefully end the interpreter loop.

L. Example Queries

Set up the sample database as in Part I. Then try the following:

```
/* TO TEST PROJECTION */
I = PROJECT INTEREST FROM INTERESTS;
PRINT I;

/* TO TEST SELECTION */
OFF1 = SELECT OFFERING
        WHERE CNUM >= CSCI200 AND
              CNUM <= CSCI40O AND
              START_HOUR = 9:00 AND
              DAYS = MW;
OFF2 = SELECT OFFERING
        WHERE INSTRUCTOR = DENEKE OR
              INSTRUCTOR = BURRIS AND
              START_HOUR = 8:00;
PRINT OFF1, OFF2;

/* TO TEST JOIN */
DEPT_INT = JOIN DEPT, INTERESTS ON NAME = NAME;
PRINT DEPT_INT;

/* JOINS, PROJECTS, and SELECTS can be nested - they return tables */

/* Try the following queries: */
Find all courses taught by a given instructor.
```

```
Find all faculty interested in 'AI' or 'DBMS'.
Find all faculty and the section numbers of the courses they teach.
Find all times between 8 to 5 on MTWR when no faculty teach this semester.
Find all prerequisites for a given course.
```

III. Extensions to SURLY

The following are extensions of basic SURLY, beyond the scope of Assignments I, II and III. You should read and understand this section but you are not required to implement any of these commands.

A. VIEW

```
command ::= VIEW viewname = relexpr END_VIEW;
relname ::= relationname | indexname | viewname
```

The VIEW command allows a user to create a "virtual" relation by specifying how to build the relation without actually building it. The relation is virtual in the sense that it does not actually exist. though it appears to exist to the user. The user does see a difference between real and virtual relations though in that virtual relations cannot be updated – since views don't actually exist, insertion and deletion are undefined operations on them. Views do have an advantage over explicit relations in that they are always up to date. Consider an EMPLOYEES relation and

```
VIEW FEMALE_EMPLOYEES = SELECT EMPLOYEES WHERE SEX = FEMALE; END_VIEW;
```

Now if we insert a few more female employees into EMPLOYEE and PRINT-FEMALE EMPLOYEES; an up to date view of the FEMALE EMPLOYEES is the result. It we had just used the relation assignment statement FEMALE_EMPLOYEES = SELICT EMPLOYEES WHERE SEX = FEMALE; then inserted the new female employees into EMPLOYEES. And then printed FEMALE_EMPLOYEES. the result would not have included the new employees. The binding time of a view is immediate. whereas derived relations (real. non-base relations) may have been bound arbitrarily many operations ago.

The mechanism whereby views are implemented is very simple. and might be called "deferred evaluation". Rather than evaluating the relexpr when the view is created, the source string representing the relexpr is saved. This string is evaluated whenever the associated viewname appears in a PRINT or a relation assignment statement. The DESTROY command can destroy views.

B. TRIGGER

```
command ::= TRIGGER relationname ON operation commands END TRIGGER;
operation ::= INSERT | DELETE
```

Derived relations are efficient in that they are not constantly recreated, but they may easily become out of date if the underlying base relations are modified. Views are up-to-date but are expensive to re-create. The TRIGGER command gives the user the advantages of both derived relations and views. A trigger is a sequence of commands to be invoked whenever some operation on a relation occurs. For instance, given the base relation BUY (buyer, thing-sold.

seller, time), and the derived relation OWN (person, thing), a trigger can be defined to automatically update the OWN relation whenever an insertion is made into the BUY relation:

```
TRIGGER BUY ON INSERT
   DELETE OWN WHERE PERSON = $SELLER$ AND THING = $THINGSOLD$;
   INSERT OWN $BUYER$ $THINGSOLD$;
END_TRIGGER;
```

Now if "INSERT BUY JOHN CAR5l MARY 8-3-79;" is executed, the trigger automatically deletes the fact that Mary owns car5l and inserts the fact that now John owns it.

Probably the best way to implement triggers is to store the trigger command body as a string associated with the triggered relation. Modify INSERT and DELETE to check for triggers when inserting/deleting a tuple into a relation. The $attrib$ notation is used in triggers to mean: "replace $attrib$ by the corresponding tuple field before executing the code". A relation need have only one trigger (of arbitrary complexity) for insertion and one for deletion, so to destroy a trigger, replace it by a null trigger:

```
TRIGGER BUY ON INSERT '' END TRIGGER.
```

C. INTEGRITY CONSTRAINT

```
command ::= INTEGRITY_CONSTRAINT relationname WHERE (qualifier);
```

The INTEGRITY_CONSTRAINT command associates with a relation a membership test (the qualification) that new tuples must pass before they can be inserted. For example:

```
INTEGRITY_CONSTRAINT INSTRUCTOR WHERE (IAGE > 17 AND IAGE < 75);
```

only allows a user to create new instructor tuples when the instructor's age is in a legal range -- tuples like <Thompson 92> should be printed as errors and should not be inserted. To implement the INTEGRITY_CONSTRAINT command, store the qualification with the relation in the RELATION table and change INSERT to check for and evaluate a qualification if it's present, before inserting the tuple. Note: there may be several integrity statements per relation. If you want to support destroying integrity statements, then add a name field to name each integrity statement.

D. RETRIEVE – a calculus operator for SURLY

```
command          ::= RANGE tuplevar IS relationname;
relexpr          ::= RETRIEVE (<rel attrib>) WHERE (qual)
tuplevar         ::= identifier
relattrib        ::= tuplevar.attrib
qual             ::= cf | cf AND qual | cf OR qual
cf               ::= relattrib relop value |
                     relattrib = relattrib
```

Example: Given relations COURSE, OFFERING, and STAFF from Figure 2:

```
PRINT Tl =(RETRIEVE (STAFF.RANK,COURSE.TITLE)
              WHERE (COURSE.CNUM = OFFERING.CNUM AND
                     OFFERING.CNUM < CSCI5OO AND
                     OFFERING.INSTRUCTOR = STAFF.NAME)
            GIVING (RANK,COURSE_TITLE));
```

```
EQUAL? (T1, PROJECT (JOIN (JOIN COURSE
                             AND SELECT OFFERING WHERE (CNUM < CSCI5OO)
                           OVER CNUM AND CNUM)
                     AND STAFF
                    OVER INSTRUCTOR AND NAME)
             OVER (RANK,TITLE)
             GIVING (RANK,COURSE_TITLE));
```

would print

T1	
RANK_	**COURSE_TITLE**
…	…

T2
TRUTH_VAL
T

Description. Note from the above example, in the qualification,

```
relattrib = relattrib      is a JOIN
relattrib relop value      is a SELECT
RETRIEVE(<rel attrib>)     is a PROJECT
AND                        is similar to INTERSECTION
OR                         is similar to UNION
```

Before implementing RETRIEVE, read the chapter on query optimization. Providing the user with both an algebra and a calculus query ability is really a more user-oriented so1ution to the usual problem of which sort of query language is best for the user.

An interesting extension to SURLY relating to RETRIEVE might be to translate RETRIEVE into a sequence of SELECT, PROJECT, JOIN, ...'s and then allow the user to examine the translation.

E. Hierarchical Formatted PRINT

One problem with relational databases is that a database consists of a flat collection of tables. In hierarchical databases, data is stored in tree form and viewed by the user in tree form. Several well known problems occur when data is stored in trees --the insert/delete/update anomalies, data redundancy, query asymmetry --and these problems provide the motivation behind Codd's normal forms which require that domains of relations be simple or atomic and not structured (subtrees).[5]

[5] Note: XPATH and XQUERY are modern query languages for querying tree-structured XML data.

Nevertheless, hierarchies are a very natural way to view some kinds of data. Because hierarchies are a useful way to view data, an extension to SURLY's PRINT command provides a formatted way to print hierarchies of relations. Figure 5 shows the form of the output for a hierarchical PRINT command given the sample data shown.

```
   command               ::= PRINT (<printobj>) FORMAT (<fspec>);
   printobj              ::= relexpr | hspec | value
   hspec                 ::= relname (<attribspec>)
                               [WHERE (qualifier)]
                               [KEY (<attrib [direction]>)]
   attribspec            ::= attrib | value | INDEX |
                             attriblist = attriblist OF hspec
   fspec                 ::= A[(length)] | I(length) |
                             SKIP[(length)] | X[(length)] |
                             (fspec) | n fspec
```

The following algorithm approximates how a hierarchical print can be interpreted.

```
PRINTSPEC (hspec, format of the form(fspec), level initially 1) RECURSIVE;
   1) TEMP = relationname from hspec
   2) if the WHERE qualifier is present
      then TEMP = SELECT relname WHERE (qualifier)
   3) if the KEY clause is present
      then TEMP = order TEMP by the KEY
   4) for each tuple in TEMP (letting INDEX (current hspec level) vary
      from 1 to CARDINALITY(TEMP)) do:
         CASE next symbol is a
            attrib   --print the value of attrib (current tuple-id)
            value    --print the value (strings must be in quotes)
            INDEX    --print INDEX(level)
            attriblist = attriblist2 of hspec2 --a JOIN-like recursive step
               PRINTSPEC (Select from relationname2 just those tuples
               whose attriblist2 JOIN fields match attriblistl JOIN fields,
               FORMAT = next unlimited repeating group,
               level -= level + 1)
END PRINTSPEC
```

Good old recursion! Do you see the SELECT, PROJECT and JOIN-like operations that make up this PRINT?

A note about interpreting FORMATS: Since hspec's are variable length, formats for them are specified using 'unlimited repeating groups.' When an hspec has been printed, the current unlimited repeating group is exited.

INSTRUCTOR	
NAME	**AGE**
Deneke	30
Evans	30
Hughes	20

COURSE OFFERING			
INAME	**TIME**	**PLACE**	**CNUM**
Deneke	MTWF12	CF314	CSCI330
Deneke	MWF2	CF316	CSCI305
Evans	TR11	A103	CSCI451
Hughes	TR5	A102	CSCI605

COURSE	
CNUM	**DESCR**
CSCI330	‘DBMS1’
CSCI305	‘Algorithms’
CSCI241	‘Data Structures’

```
PRINT ("COURSE OFFERING BY AGE BY PROFESSOR
        INSTRUCTOR ('AGE  -- ', AGE, NAME = INAME OF
            COURSE_OFFERING (INDEX, INAME, CNUM = CNUM OF
                COURSE ('COURSE=', CNUM, 'DESCRIPTION=', DESCR))
            WHERE (CNUM < 'CSCI600')
            ORDER (INAME ASCENDING, CNUM DESCENDING))
        ORDER (AGE ASCENDING))
FORMAT (A, SKIP,                                  <= title
         (SKIP, A, I(3),                          <= 'age—', age
            SKIP, I(3), X(3), A,                  <= index, instructor's name
              SKIP, X(8) 2 A, X(5), 2 A))));      <= 'course=', cnum,
                                                     'description=', descr
```

prints

```
COURSE OFFERING BY AGE BY PROFESSOR

AGE  -- 20
  1     HUGHES            <= no tuples where CNUM < 'CSCI600'
AGE  -- 30
```

```
1     EVANS
        COURSE = CSCI241  DESCRIPTION = DATA STRUCTURES
2     DENEKE
        COURSE = CSCI305  DESCRIPTION = ALGORITHMS
        COURSE = CSCI330  DESCRIPTION = DBMS1
```

Figure 4: Sample Data, A Hierarchical, Formatted PRINT Command, and the Corresponding Output

F. LINK

```
command ::= LINK relationnamel AND relationname2
            OVER attriblist1 AND attriblist2
```

Since some relations will often be joined over join domains, it may speed things up if we maintain an 'index" which relates tuples in the first relation to the tuples they will join with in the second relation. To this end, implement the LINK command. You will have to modify JOIN to take advantage of this new associative index. Also, be sure to update the LINK when INSERT's and DELETE's occur. How does LINK compare with the DBTG SET? Links are non-information bearing or automatic associations (that is, like indices, they are there for efficiency and add no new information).

G. INDEX Revisited

Modify the SURLY index statement to:

```
INDEX indexname ON relationname
                ORDER (attrib [direction] {attrib [direction]})
                STORAGE STRUCTURE storage structure;
```

where

```
direction          ::= ASCENDING | DESCENDING
storagestructure   ::= HEAP | TREE | HASH [(tablesize)]
```

If the storage structure is TREE, the "direction" of ORDER attributes may now be given (the default is ASCENDING); if the storage structure is HASH, the user may specify the size of the primary hash table for this index (or it may be defaulted to, say 25). This will involve dynamic memory management.

H. B-TREE

Replace binary trees by B-trees as SURLY's TREE storage structure. This change is transparent to the SUR user (physical data independence) except that it may speed up access.

I. LOG, UNDO, DUMP, RESTORE

Implement a LOG of all SURLY transactions (commands) recording what tuples are inserted/ deleted for a given command (INSERT's, DELETE's, INPUT's). Try to figure out how to implement an UNDO statement which has the effect of restoring a SUR database to a previous state. For example:

```
DELETE STAFF WHERE RANK = GTA;
PRINT STAFF;
UNDO;
```

Here, all GTA tuples are deleted from the STAFF relation, then the relation is printed, and finally the relation is restored to its previous state. UNDO should optionally print the transaction being "undone"; and the user should be able to call it several times in a row (UNDO; UNDO; ...). , .

Periodically dump/export the data base (RELATION, INDEX, NAME_SPACE, LIST_SPACE, and any system variables) along with the current log. A new log should be started. A measure of how often to dump the database might be in terms of the number of operations (INSERT's and DELETE's) since the last dump. Implement this second method by allowing the database user to use the "DUMP [(number-of-operations)];" command (DUMP aka EXPORT).

If "number-of-operations" is not present, the database is dumped immediately. If "number-of-operations" is present, then dump the database automatically after the specified number of operations. Be sure to print a message telling, the user that a DATABASE DUMP has occurred (is occurring). Also implement a restore operation which reads a database dump file and restores the database to its state at the time the dump occurred.

The DUMP/EXPORT and RESTORE/IMPORT operations should be useful to anyone wishing to save the current state of the world at one point, then return to it at some later point (like after lunch, for an interactive SURLY user).

J. READ/WRITE

```
command  ::= READ filename;
command  ::= WRITE filename <relation>;
filename ::= identifier implementation dependent
```

Description. READ reads a file formatted as a sequence of SURLY commands. WRITE writes a set of <RELATION descriptor, INDEX descriptor(s), INPUT-tuple descriptor> onto the specified file in a form that can be read by READ.

If DUMP is implemented by READ and RESTORE by WRITE (instead of saving NAME_SPACE and POINTER_SPACE), then a user can gain two benefits:

1) A DUMP can clear NAME_SPACE, which can become clogged with strings from relations not in use.- So another measure of when. to DUMP is when NAME_SPACE is getting full.
2) If DELETE WHERE works by marking deleted tuples, then this wasted space can be reclaimed if POINTER_SPACE is re-initialized after a DUMP.