OhBench: The Ozsoyoglu-Hou workbench for database experimentation

Shashi K. Gadia
Computer Science Department
Iowa State University
Ames, IA 50011
gadia@cs.iastate.edu

Abstract. OhBench is a workbench for database system development. We have named it after Gultekin Ozsoyoglu and Wen-chi Hou, who designed and implemented it at Case Western Reserve University. The original name of the workbench was ERAM.

The target of the workbench is to deliver OhBase, a general purpose database system, which can be used by a user community to program database applications in an SQL-like multiuser environment. The design of the workbench is open ended, leaving ample of possibilities for changes and extensions. OhBench includes the source code and it is suitable for interesting programming projects in a realistic time frame. In this document we give a brief account of OhBench and OhBase.

Acknowledgments. OhBench was generously provided to us by Gultekin Ozsoyoglu few years ago. It was substantially documented by Gautam Bhargava. Sunil Nair headed a team which implemented a temporal database system, including a temporal query language, storage structures for tuples and routines to input and output information. Sunil Nair and Simon Cheng have maintained it when it was used for projects in a graduate level database course Com S 561 and Com S 661 at Iowa State University. It has been used for numerous projects in these database courses. Kenneth Allen, Chun-Hsien Chen and Tsz S. Cheng have used it as the main platform for database projects for their M.S. Creative Components at Iowa State University. We thank all these people who have contributed to better understanding of the system.

Caution. A developer is cautioned that from time to time this document may become out of synchronization with OhBench and OhBase.

Contents

Appendices

Following appendices are considered an integral part of this document, although they are not physically included here. Each appendix is a text file stored in the OhBench directory, and most of them are active files in OhBench As a reminder of its importance, appendices also appears at the end of this document, where the file names are provided.

1. Introduction

In this section we introduce terminology used in this document. We give a brief outline of the organization of the directories for OhBase and OhBench. We also discuss how OhBench and OhBase may be installed. Finally we discuss how to track development of a project.

We begin by introducing some terminology to help us make clear distinction between roles of different users and personnel.

- A *developer* is one who uses and modifies the workbench to affect a change in OhBase. The term user will only refer to a user of OhBase and never to a developer. We assume that a developer has full access to the workbench and the unix system on which the development is taking place. From the point of view of development, we do not identify any user hierarchy.
- We recognize the role of the *Unix Administrator* in the context of an installation where OhBase is running. The unix administrator authorizes the usage of OhBase. The authorized user is either a *superuser* or *enduser*.
- A *superuser* can create one or more database applications, and becomes the *owner* of such applications.
- The owner of a database application can enroll endusers for the application. An end user authorized to use a database application has access to the tables created by the owner. In addition the end user can create one or more relations and become the *owner* of such relations; these relations are not available to other users.

Note that a unix user can be an owner of some databases and enduser of other databases. The unix administrator keeps track of users and their authorization status with different databases.

Organization of OhBase

OhBase is organized as a unix directory as shown in Figure 1(a). The bin subdirectory contains modules to allow a user to interact with the OhBase. OhBase includes a central catalogue (dbCat) of all database applications, a central directory (dbUsers) of all users and their privileges, and a central storage (dbStore) for all database applications. In addition OhBase contains a demo database together with example of a user session. It also contains unix style user manuals for documenting the OhBase dbms.

Organization of OhBench

OhBench is organized as a unix directory as shown in Figure 1(b). It includes lexical analyzer, grammar, and source code for OhBase. The lexical analyzer and grammar are inputs to the unix compiler-compiler utilities *lex* and *yacc* to generate a compiler for OhBase. The source code for OhBase is also stored here. *Makefile* and *MakeDB* utilities which are used to express dependencies among modules and affect incremental compilation. The modified or new modules are also stored in the OhBench directory. *MakeDB* also takes care of copying the necessary object modules to the OhBase directory and linking. In addition there is an *install.bat* file and a *ProjectTracking* directory. Their role is explained below.

OhBase
- *bin* (executable code)
  - *createdb* (modules to create a database)
  - *deletedb* (modules to delete a database)
  - *listdb* (module to list all database schemes)
  - *loaddb* (modules to load a single database)
- *dbCat* (catalog of all databases)
- *dbUsers* (directory of users and privileges)
- *dbStore* (data store)
- *demo* (demo database and modules)
- *man* (OhBase manuals)

(a) OhBench directory

OhBench
- source code files
- .o files
- *typedef*
- *clex (File containg lexical rules)*
- *cgrammar (The grammar file)*
- *Makefile*
- *MakeDB* (Command sequence to generate OhBase)
- *UtilsAndInfo*
  - *Install.bat* (Install OhBench + OBase)
  - *ISession.txt* (A sample installation session)
  - *DSession.txt* (sample OhBase demo session)
- *ProjectTracking*
  - *report.txt*
  - *addedFiles*
  - *modifiedFiles*
  - *delatedFiles*
- *Temporal* (A temporal database: ignore it)

(b) OhBase directory

Figure 1 OhBase and OhBench directories

Installation of OhBench and OhBase

A personal copy of OhBench and OhBase can be made by executing install.bat from the directory where you want to install OhBench and OhBase. In addition this utility also enrolls you as a enduser of "demo1" database. A sample of the activity you should expect to see during your installation

session is shown in the file *InstallSession.txt*. A copy of these files appear in appendices.

Tracking a project development

To keep track of the incremental development in a project, a directory *ProjectTracking* has been created. Initially this directory contains an empty *readme.txt* file and empty subdirectories *addedFiles*, *modifiedFiles*, and *deletedFiles*. It is suggested that

- New routines are stored in *addedFiles.*
- The existing routines which are modified are stored in *modifiedFiles.*
- The existing routines which become unnecessary are placed in *deletedFiles*.
- A report of a project is stored in *readme.txt* file in the *project* directory. The final destination of the files in the above three subdirectories should be suggested, but should not be carried out.

In the remainder of this document we discuss some of the important components of OhBase and OhBench.

2. User interface in OhBase

The executable modules are stored in the *bin* subdirectory in OhBase. OhBase provides two level of commands. From the unix level, database applications can be created, destroyed, and listed. A single database application can also be invoked from this level entering the application level.

The following commands can be issued from the unix operating system level.

- *createdb* dbname
- *deletedb* dbname
- *listdb*
- *loaddb* dbname

To execute the first *createdb*, *deletedb* and *listdb* commands the user should be registered in the dbU*sers* file as a superuser. The command *createdb dbname* creates a database application named *dbname*. The user who executes this command becomes the *owner* of *dbname*. The *deletedb* command can be used to delete an application from the OhBase system. The *listdb* command can be used to list all the database applications stored in the OhBase system.

Database maintenance

The *loaddb dbname* starts an interactive session for the *dbname* application, and the OhBase system prompt becomes @. To use this command the user must be registered as a user of *dbname*. The following commands can be issued at the @ prompt.

- *create* to create a relation in the database
- *scheme* to print scheme of a relation
- *copy* to move data between a relation and a file
- *append* to append tuples to an existing relation
- *update* to update value of tuples
- *print* to print contents of a relation
- *delete* to delete tuples from a relation
- *destroy* to destroy existing relations(s)
- *rename* to change names of attributes in a relation

Querying a database

The query language of OhBase is algebraic. The available operators are: *union*, *diff*, *project* (to remove and rearrange attributes), *njoin*, *cprod*, *rename*, and SQL-like *select ... from ... where* statement. An example of a query is given below.

```
@select Name, Salary
  from emp union new-emp
  where Dept = "Toys"
```

When a query is executed, OhBase responds by displaying the result on the screen.

The following operators are also available but we omit them: *setf*, *pack*, *unpack* and operators for aggregation.

Documentation, help and a user session

The unix style user manuals are stored in *man* subdirectory. *Help* command is also available at the @ prompt. In addition, demo database application is available in the *demo* subdirectory. See Appendix 14 for further references, and Appendix A for example of a user session in OhBase.

3. The database catalog (dbCat)

A catalog of all database applications in the OhBase system is stored in a tree like structure as shown in Figure 2(a). The tree is stored in the file *dbCat*. The root contains an entry for each database. A database entry points to a node which contains entries for all relations in the database. Each relation entry points to a node which contains entries for all attributes in the relation. Database, relation and attribute entries are discussed shortly. We first discuss the physical structure of the catalog.

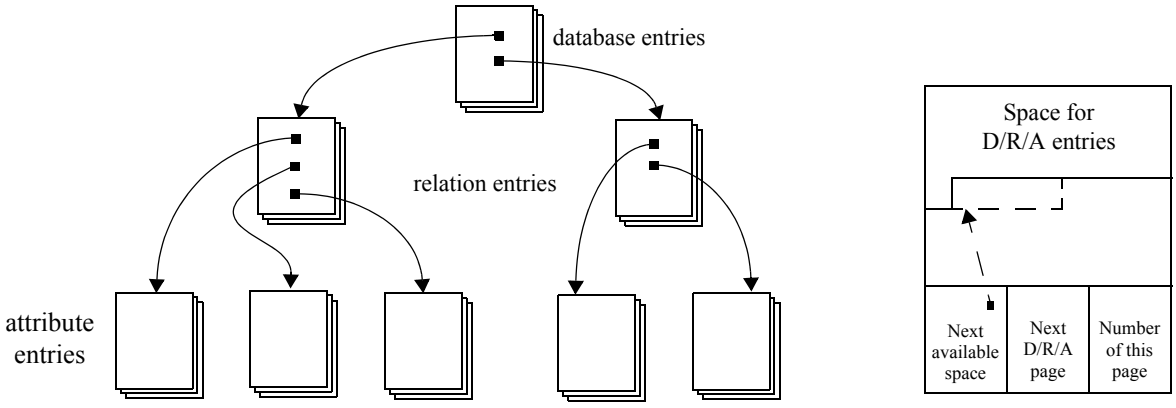
Physically, the *dbCat* file consists of pages. A given page is of one of the following types: a database page (D), a relation page (R), or an attribute page (A). A node in the tree can contain one to more page of the same type. The structure of a page in the file is shown in Figure 2(b).

Entries of a given type (D/R/A) are entered one after another starting from the beginning of a page. These entries will be discussed later in this section. The following information is stored at the end of each page.

- **Next available space.** A pointer to the beginning of available space in the page.
- **Next page.** Page number of the next page in the same node of the catalog. Note that all pages in a given node are of the same type.
- **Page number.** The sequence number of the page in the file.

The database entries.

A database entry consists of a database name and pointer to the node which contains relation entries of all relations in the database.



(a) Database catalog organized as a tree: Each node is a sequence of pages

(b) Page layout

Figure 2 Architecture of database catalog (dbCat)

The relation entries.

A relation entry consists of the following:

- **Flag.** The flag is set when the relation represented by this tuple is deleted. The space occupied by this tuple and its corresponding tuples in the attributes relation can be reclaimed later on by OhBase garbage collector.
- **Relation-name.** A relation is stored as an alphanumeric string of up to 8 characters.
- **Owner-id.** The *unix id* of the relation owner. Note that to create a relation, user does not have to be the owner of the database. When a user creates a relation, the OhBase system creates a unique id by concatenating unix *id* of the user and the relation name. This becomes the unix file name of the relation being created.
- **Number of attributes.** The number of attributes in the relation. (Current limit is up to 80 attributes).
- **Keyno1-keyno6.** The six fields contains the attribute numbers which comprise the primary key.
- **Organization.** The storage structure for the relation. Currently only heap structure is available, and it is indicated by the default value “organization = 1”.
- **Compression.** Describes if any compression technique is used. Currently no compression is available.
- **Number of tuples.** The total number of tuples in the relation being represented.
- **Index-page.** The total number of pages in the primary index file for the relation being represented. Note that currently the index structure is an array of key, tuple locations pairs. The array is sorted by key, on which a binary search is performed. In an OhBase relation one cannot insert two tuples with the same key values.
- **Atr-pointer.** A pointer (page number) to the first page of attributes relation for the relation being represented.
- **Reserved space.** Space reserved for later expansion

The attribute entries.

An attribute entry contains the following fields.

- **Attribute name.** An alphanumeric string of up to 8 characters.
- **Attribute number.** The sequence number of the attribute in the relation scheme.
- **Attribute type.** An attribute type is either atomic or nested.
- **Datatype.** Data types allowed are:
  - character strings (up to 256 bytes)
  - integer (2 byte or 4 byte)

- real (double precision)

- **Bucket size.** The number of instances that can fit in one bucket. A bucket is a block of space allocated for the instances of a set-valued attribute.

- **Attribute size.** The size of the attribute in bytes, not exceeding 255 bytes.

- **Format.** This is for I/O. This consists of a "%" followed by the length of a data type in the attribute, e.g. %5c or %2i.

4. Directory of users (dbUsers)

The directory of users who are allowed to use OhBase is stored in the file *dbUsers*. This file is a line oriented text file and it is only accessible to the unix administrator. Each line contains information about a single user. The information consists of the following fields separated by ";".

- **User name.** The unix user name.

- **User ID.** The unix user ID.

- **Capability.** This field is "1" for the superusers, and "0" for endusers. The terms superuser and end user are with respect to a given database application. A superuser is allowed to create one or more database applications, and becomes their owner. An application has only one owner, but a superuser can be the owner of several applications. An enduser user is identified as "0". Such a user cannot create a database application. An owner can create relations for the all end users of the database. An enduser can create additional relations, but such relations are private to that user. Note that a superuser can be an enduser of database not owned by him/her.

- **Database names**. This field contains the names of all database applications which may be accessed by the user. The database names are separated by ",".

5. The data store

The data consists of relations and indices. These are stored in dbStore subdirectory. Figure 3 shows the structure of a page. Note that the page supports variable size tuples. Tuples can be marked for deletion. Garbage collector is internally invoked by OhBase to claims all the free space due to tuple marked for deletions.

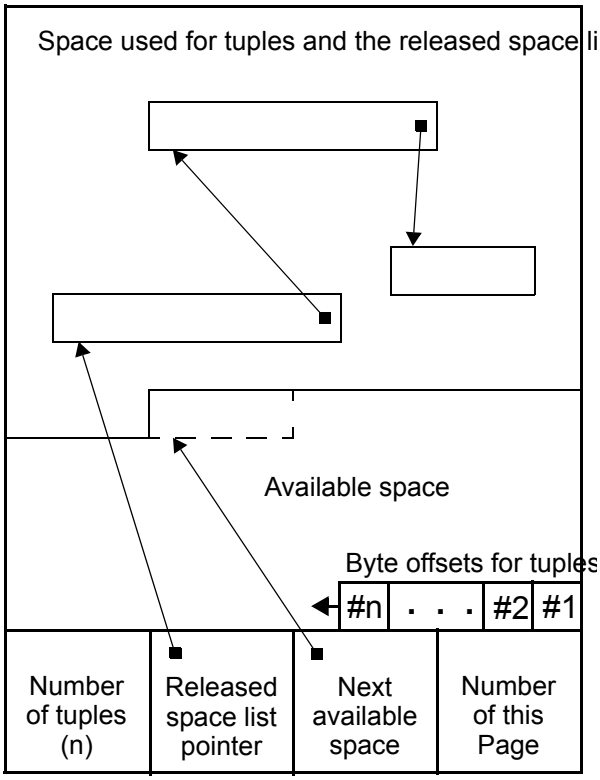


Figure 3 Page format for relations in dbStore

6. Relation descriptor

Relation descriptor is an important, and perhaps the most widely used main memory resident data structure in OhBase. Whenever a relation is opened, a relation descriptor is allocated for it. When a relation is closed, the relation descriptor is deallocated. The relational descriptor for a relation consists of the following:

- The information from the database catalog about the relations in use

- A formatted buffer containing the tuple being processed currently

- pointers, called the *file pointers*, to the data structures (defined in the unix standard I/O library) for the opened files (i.e. the relation and the related primary index file). The data structure contains several pieces of information about a file:
  - A pointer to the buffer (1024 bytes).
  - A pointer to the next character position in the buffer.
  - Some flags describing read and write mode.
  - Some constants.

- Some temporary variables such as the current position of the file, tuple identifier, the total number of

tuples in the file buffer, and the position of the bucket or a tuple in the file buffer.

7. Access methods

For tuples of a relation two access methods are provided.

- **Sequential.** Accesses a relation tuple by tuple.
- **Indexed.** First obtains the tuple identifier from the primary index file, and then gets the tuple from the relation.

8. OhBase software system structure

As shown in Figure 4, the OhBase software system consists of four modules. At the top level we have the *command interpreter*. The *relational maintenance module* and the *algebra module* are at the same level. The lowest level module is the *file management system* (FMS).

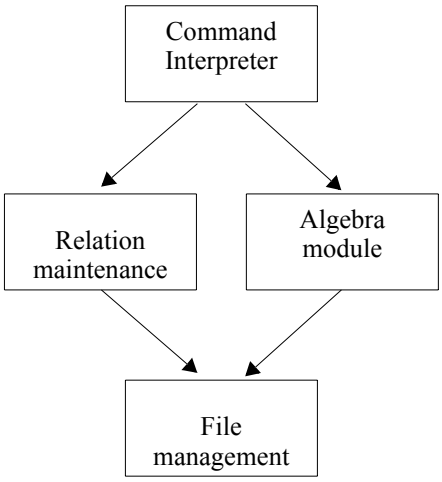


Figure 4 OhBase Software system architecture

The command interpreter

The command interpreter module is the highest level module. Users interact with OhBase at this level. OhBase uses the compiler module [yylex + yyparse] to parse user commands. (See Figure 5.) The result of the parser is a parse tree. Parse tree in OhBase is discussed in more detail in Section 9.

The relation maintenance module

The modules includes the functions for all the database maintenance operations. These functions are performed eventually by calling the routines in FMS.

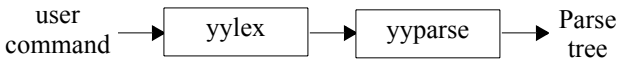


Figure 5 Parsing user commands

The algebra module

The algebraic operators include union, difference, cartesian product, select, project, pack, unpack, aggregate by template, natural join, etc. The output of a relation can be displayed, stored, or be operand to another operation. If it is stored, its name should not previously exist in the catalog.

File management system (FMS)

The file management system (FMS) is the lowest level module in the system. It is invoked by higher level modules when required. FMS is used for reading and writing tuples and garbage collection in a relation. The following is a list of functions provided by the FMS.

- **Read/write catalog (readdict, newrel).** The routines *readdict* and *newrel* are used to read from and add tuples to the catalog
- **Open and close relations (ropen, rclose).** The routine *ropen* allocates memory for a relation descriptor, fill it with the related information in RELATIONS relations and ATTRIBUTES relation, opens the files for the relation and its primary index file if it exists and initializes some temporary variables. The routine `rclose` deallocates all the memory taken by the relation descriptor and the file buffer(s). The files for the relation and its primary index files are closed. If the relation is temporary (e.g. an intermediate relation), it is removed.
- read a tuple **(sread, dread)**. These routines read a tuple using the corresponding access methods. According to the datatype and the size of each attribute, from the file buffer to the relation descriptor. If there is no tuple left in the file buffer, the next page is read into the buffer from the disk (when sequential access is used).
- **Garbage collection (release, getspace).** Space released from tuples is inserted into the released-space list in each page, and consecutive unused segments are collapsed together into larger segments. When space is needed again, the first fit method is used to get the requested space.

```
struct node
{int ndtype;
 char *namep;
 union
  {int ival;
   double fval;
  }uval;
 struct node *lf;
 struct node *rt;
};
```

ndtype	*namep*
uval	
lf	*rf*

The *uval* field is not always displayed.

(a) C declaration of node (b) A graphic for a node

Figure 6 A node in parse tree

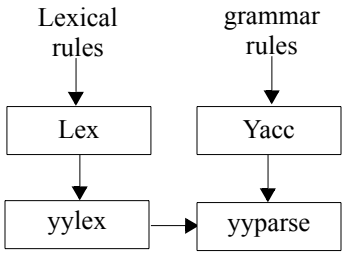


Figure 9 Creating parser with Lex and Yacc

- **Name (namep).** In some cases this string simply restates the node type in words, and thus it is redundant, but helpful in debugging. In other cases it stores more useful information such as name of a variable in the user query.
- **Value (uval).** This field is used only by a node representing a constant in the user query. The type of constant is stored in the ndtype field, and the constant is stored in the namep field as a string. In case the constant is intended to be numeric, its value is computed and stored in this field.

  As uval field is rarely used and has little to do with the structure of a parse tree, it is sometimes not exhibited.
- **Child pointers (lf and rt).** In case the node represents a binary operator, the two child pointers point to the two operands. In case of a unary operator, the left pointer (lf) points to the operand, and right pointer (rt) points to additional information, such as, the condition in case of a selection, and attribute list in case of a projection.

9. Parse tree and its evaluation

As shown in Figure 5, a command is represented as a binary tree internal to the workbench. The parse tree is built recursively in terms of the components of a command. The internal structures for representation of algebraic operators are shown in Figure 7.

Nodes in the parse tree

A node in the tree is declared as a C structure containing five fields. (See Figure 6.) The details of these five fields is as follows.

- **Node type (ndtype).** This integer indicates the type of node, e.g. a relational operator, a binary comparison, a variable, a constant. Some possible values, e.g. UNION, are declared as global constants.

Node allocation

The following routine in OhBase creates a new node, and returns the pointer to the node.

```
  struct node *newnode(type,nmp,lfp,rtp)
int type;
char *nmp;
struct node *lfp,*rtp;
{    char *cp;
     struct node *np;
     np= (struct node *)
     calloc (1,sizeof (struct node));
     if((cp = (char *)
     calloc((strlen(nmp)+2), sizeof(char)))
     == NULL)
  {  fprintf(stderr,"newnode:
     no memory available \n");
     exit(0);}np->ndtype= type;
     strcpy(cp, nmp);
     np->namep = cp;
     np->lf = lfp;
     np->rt = rtp;
return(np);
}
```

An example of a parse tree is shown in Figure 8. The parse tree is evaluated bottom up, i.e. from leaves toward the root. As stated above, ndtype, the first field in a node, says what type of node it is. This helps simplify the logic for evaluation of a parse tree. The node type of the root is ALGEB, a global constant. This node indicates that the expression tree has been terminated and it can be destroyed. The only child of the root is PRINT, another global constant, which forces the result of the query to be displayed to the user on a terminal. This node is followed by the parse tree for the actual query.

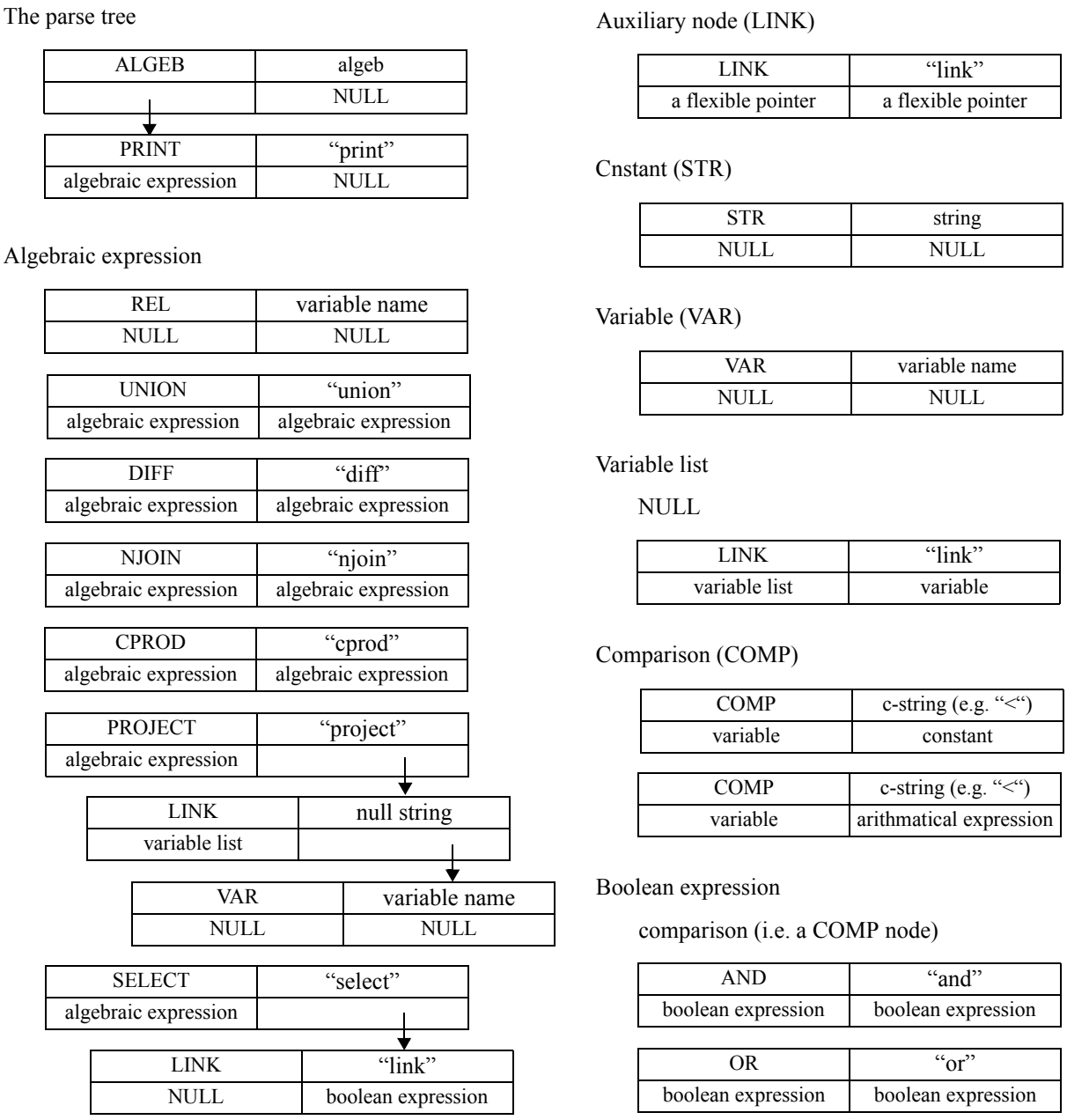


Figure 7 Representation of algebraic expressions

10. OhBench development environment

The source code, lexical rules and grammar are stored in the *OhBench* directory. In addition, labeled command sequences are stored in *Makefile* for incremental compilation of .c files. Selected labels from Makefile can be stored in the executable file *MakeDB* for incremental compilation.

Compiler generation

The SQL-like language can be modified, or a different language can be implemented by a developer using the unix utilities *lex* and *yacc*. First, the lexical rules and grammar for the language should be provided in a form acceptable to *lex* and *yacc*. Then the parser is generated by calling *lex* and *yacc* to generate a compiler for the language. Lex and the compiler-compiler *yacc* (yet another compiler compiler) are very interesting and to use them one does not need expertise in compilers. Appendices D contains the lexical rules and Appendix E

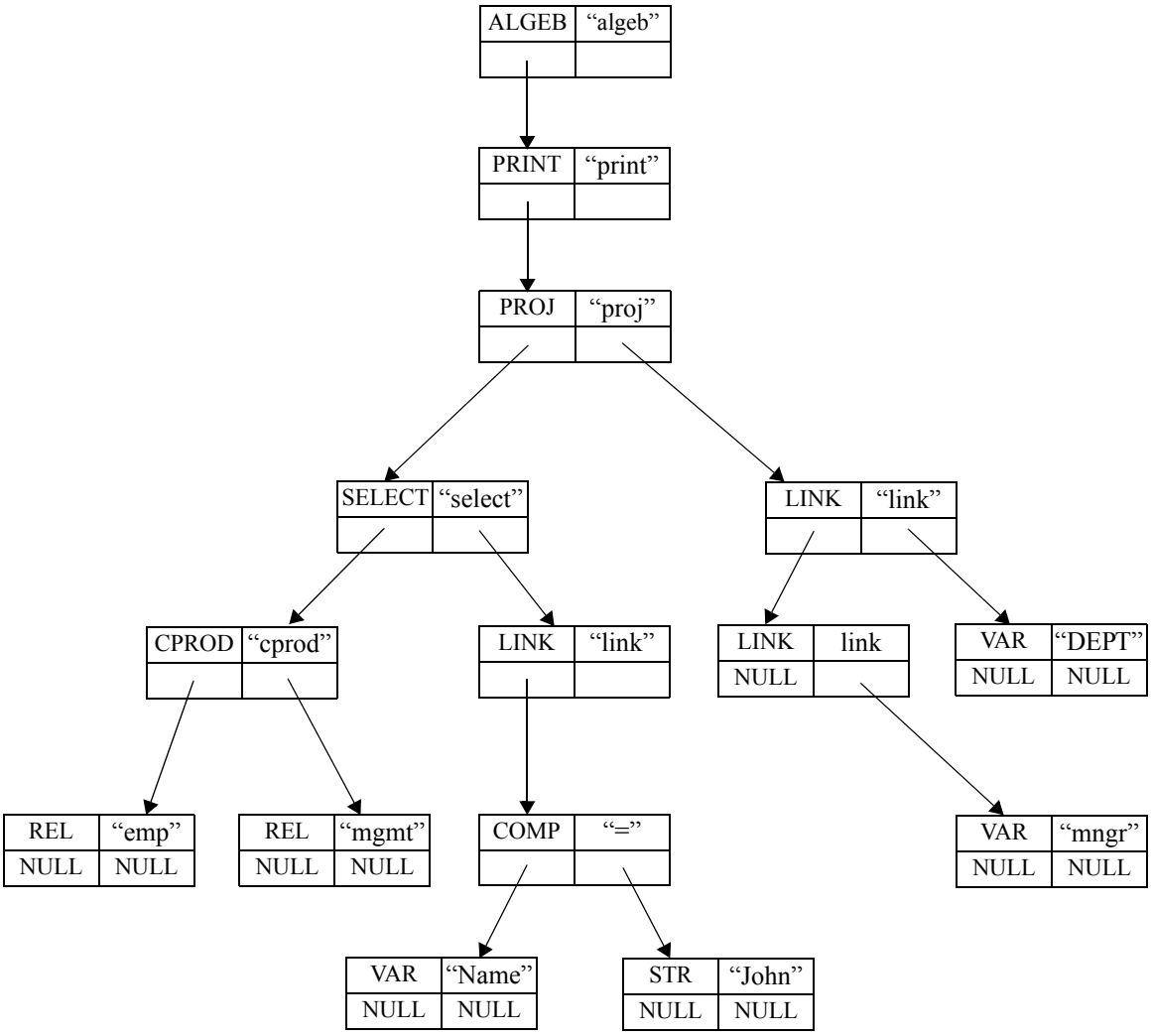


Figure 8 Parse tree for
project (*select from* (emp *cprod* mgmnt) *where* (Name = "John")) on "Dept Mngr"

contains the grammar used for OhBase. Note that these appendices are given to make this document more self contained; the actual contents of these modules may be different.

11. Example of a project

In this sample project we hint how you may proceed to implement a simple algebraic query optimizer in OhBase. The optimizer uses the following algebraic identities, transforming occurrences of left hand side in an expression to those on right hand side.

- $\sigma_F(e_1 \cup e_2) = \sigma_F(e_1) \cup \sigma_F(e_2)$
- $\sigma_F(e_1 - e_2) = \sigma_F(e_1) - \sigma_F(e_2)$
- $\sigma_{F1}(\sigma_{F2}(e) = \sigma_{F1 \wedge F2}(e)$
- $\Pi_X(\Pi_Y(e)) = \Pi_X(e)$ if X is contained in Y.
- $\Pi_X(e_1 \cup e_2) = \Pi_X(e_1) \cup \Pi_X(e_2)$

The syntax for some of the algebraic operator in the query language in OhBase is as follows:

- e_1 union e_2, where schemes of e_1 and e_2 are the same.
- e_1 *diff* e_2, where schemes of e_1 and e_2 are the same.
- *project* e *on* X, where X is a list of attributes written as A B C etc.
- *select from* e *where* F, where F is a conditional expression.
- e_1 *njoin* e_2

When a user provides an input expression e, OhBase parses e, and produces an expression tree whose root is cap-

tured in the global variable called *root*. Then *root* is executed by a call *query* ( ). You need to expand it into a program segment as follows.

```
query ( )
printtree ( )
optimize ( )
printtree ( )
query ( )
```

Here, *printtree* is a routine, which prints the preorder traversal of the tree, and *optimize* changes a tree to its optimal version.

Recall that the *ndtype* field in a node is an integer represented by symbolic constants defined in the code. You can use the symbolic representation without worrying about their actual values. For example, you may use the constant SELECT as follows.

```
int i;
struct node  temp;
...
i = temp → ndtype;
if i = = SELECT
...
```

12. Some remarks about OhBase

The attributes of a relation are not stored in the parse tree. This information, available in the catalog, may be needed for further software development, e.g. algebraic optimizer. Here are some hints to solve this problem:

- The "scheme" command loads the catalog and displays information about attributes on the screen.
- The attributes of the involved relations may be made more accessible during the optimizing phase. This may be achieved by placing a dummy node in the tree just above each relation. The type of this node can be similar to the node for projection operator. At the end of the optimization phase, the dummy nodes can be removed.

There are some problems when the same attribute appears in more than one operand in a join or a cross product. The attribute name in the right operand is changed by OhBase overwriting the first character by "$". However if an attribute appears more than twice, it is not handled properly.

There are some problems associated with the keys when union and difference operators are used. First, a union or difference of two relations can be performed if the relations have the same attributes, even if the key specifications of the two relations are different. Even if both operands have the same key, problems may arise. For example, if (a,b) is a tuple in the first relation, and (a,c) is the tuple in the second operand, the union may exclude (a,c) instead of either changing the key or giving a runtime error.

In these cases the number of bytes read from or written to files may be recorded and then converted to the number of blocks required to hold the file.

To benchmark performance it would be appropriate to instrument a means of counting disk accesses. The problem is that in some cases OhBase does not fill a page and then write to memory, or read a page and then use the page until it is no longer needed. When it does not address pages as such, OhBase uses the "C" file access commands to read from and write to files, leaving the paging to unix.

13. Sample projects

1. Algebraic optimization, Part 1
2. Algebraic optimization, Part 2
3. Quel optimizer, Part 1
4. Graphical user interface
5. Database querying through Mosaic
6. Simplify the page format of a relation
7. Implement an access method
8. Port ERAM to IBM/PC
9. Extend the query language of ERAM
10. Provide support for concurrent access
11. New elementary data type, e.g. date
12. A more elaborate user hierarchy
13. Implement a temporal database system
14. Implement a multi level security database system
15. Implement a query language for a spatial database linking it to ARC/INFO

14. References

The following documents provide further details of OhBench and OhBase.

- The M.S. Thesis
- OhBase user manual
- OhBench software developers manual, and the source code

Appendix A.
A user session in OhBase

See file OhBench/UtilsAndInfo/DSession.txt

Appendix B.
Shell script to install the workbench

See fileOhBench/UtilsAndInfo/Install.bat

Appendix C.
A sample installation session

See file OhBench/UtilsAndInfo/ISession.txt

Appendix D.
Contents of Makefile utility

See file OhBench/Makefile

Appendix E.
Contents of *MakeDB* utility

See file OhBench/MakeDB

Appendix F.
Lexical rules for OhBase

See file OhBench.clex

Appendix G.
Grammar for OhBase

See file OhBench.cgrammar