University of Edinburgh
Department of Computer Science

James Clerk Maxwell Building
The King's Buildings, Edinburgh

University of St Andrews
Department of Computational Science

North Haugh
St Andrews, Fife

A DBS Architecture Supporting Coexisting User Interfaces: Description and Examples

Alan Dearle

PPR-6-83
Sept. 1983

A DBS ARCHITECTURE SUPPORTING COEXISTING USER INTERFACES: DESCRIPTION AND EXAMPLES

BY

PEDRO E. HEPP

DATA CURATOR
DOCUMENTATION

15 August 1983

Table of Contents

List of Figures

Chapter 1

Introduction

Database technology is already recognised and increasingly used in administering and organising large bodies of data and as an aid in developing software. This document is based on the author's research [Hepp 83] which considers the applicability of this technology in small but potentially expanding application environments with users of varying levels of competence. A database system (DBS) architecture with the following main characteristics is described in [Hepp 83]:

1. It is based on a set of software components that facilitates the implementation and evolution of a software development environment centered on a database.

2. It enables the implementation of different user interfaces to provide adequate perceptions of the information content of the database according to the user's competence, familiarity with the system or the complexity of the processing requirements.

3. It is oriented toward databases that require moderate resources from the computer system to start an application. Personal or small-group databases are likely to benefit most from this approach.

This document concentrates on the description of the principal components of this architecture, used to implement several user interfaces. Three of these user interfaces are described in this document: two relational query languages and a report generator. More detail and other related topics can be found in [Hepp 83], among them:

- an overview of some of the current relational database systems and their suitability for small but potentially evolving applications.
- an analysis of current software engineering problems with emphasis on rapid prototyping, software components and (internal) database standards.
- an analysis of the extended relational model (RM/T) [Codd 79] and its suitability as an (internal) database model. The supporting of functional query languages such as Daplex [Shipman 81] and FQL [Buneman 82] by RM/T is also shown.
- an analysis of the language PS-algol as a tool for writing database software. PS-algol [Atkinson 83] was the implementation language used in this research.

The availability of a language with persistent data capabilities such as in PS-algol has been an important factor in reducing software production costs.

- an analysis of relational structures in PS-algol and their impact on space and time performance.

- the results of observing and recording query language usage of more than 160 users, over a period of more than two years. These results have influenced the design of the user interfaces described in this document.

Chapter 2

The Database Architecture

2.1 Introduction

Database software is typically large and therefore expensive to acquire and maintain. However, it can be scaled down by properly identifying the levels of functionality required for smaller applications and by using adequate software development tools. It can then be cost-effective for personal or small-group databases which do not need all the functionality generally associated with large scale database software.

This chapter presents in some detail the set of components that constitutes the database architecture underlying the implemented user interfaces.

A key factor in diminishing software costs lies in the ability to reuse software components by the user interfaces. This will require the identification of common database tasks that are amenable to standardization. This can be achieved by providing an internal conceptual shema (ICS), a common internal query language (IQL) and a common pool of data manipulation routines. All of them can be provided on an incremental basis. For example, the IQL may be started with the three "basic" relational operations project, select and join, followed at a later stage by set operations, building up gradually a fully relational system. A similar approach can be defined for the data manipulation routines and the ICS structures.

Having decided on the internal data model, which in this case is the relational model, the first user interface and the initial set of components can be designed as a prototype.

2.2 A Minimal Environment

Another advantage of implementing a database architecture as a set of components is to make it possible to implement these components concurrently. The ability to test the software and its coupling to other components while writing it, requires the definition of different priorities in functionality. Some minimal set of functions that constitute a rudimentary but working system can be rapidly implemented, tested and then systematically modified and expanded. The first versions of a system can be regarded as prototype systems.

To achieve fast prototyping of the user interfaces it is desirable to design them under the principle of *minimum responsibility*, that is, they should be relieved from as much processing and decision making as possible that may distract them from their two main tasks of presenting information to users and collecting their requests. The communication with the database and the processing of requests is facilitated by providing components that accomplish each of the required tasks. For example, processing a query or ensuring the proper installation of updates should be done by specialised system components.

The principle of minimum responsibility requires though, good communications between user interfaces and the internal components. That is, there is an agreement on internal standards that are understood by the system and to which the user interfaces translate their requests and read the results.

To facilitate this communication it is desirable that user interfaces refer to objects (relations and attributes) by name instead of by system provided identifiers. This will also permit the underlying system to change the identifiers if necessary (e.g. restructuring a relation) and also allow users to define different name spaces for the same objects. Thus, a first component can be defined as a Name Handler.

2.3 The Name Handler

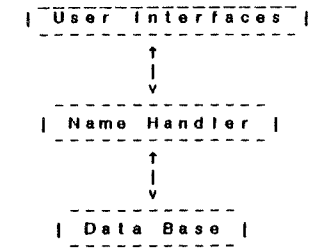
The Name Handler (NH) serves as the central node of the system. It receives all messages from the user interfaces redirecting them to the proper components as specified in the message. In that process, the NH performs a minimal compilation in translating names to internal identifiers. The existence of the NH isolates internal components from user interfaces, simplifying the internal references to objects by the use of internal identifiers and protecting the database integrity by ensuring the proper establishment of data paths. For example, attempts to modify the database by the users may be first checked for consistency and rights in a special purpose Update Handler, as directed by the Name Handler.

User interface components may regard the NH as the sole database component, providing all the required services. No other component may be visible to them. The database, as seen from the user interface components consists of the elements shown in figure 2-1. However, the possibility of having some user interfaces such as a performance analyser or data restructurer accessing the system at other levels, is retained.

For the Name Handler to translate names into internal identifiers and to ensure that the objects referred to exist, it must have access to the internal Conceptual Schema (ICS) structures. The ICS consists of a set of relations and is therefore maintained and accessed through the same data manipulation routines as any other database relation. This uniform access to both data and meta-data reduces the total cost of software and simplifies its development and maintenance by avoiding the definition of a dedicated data description language.

From the many relations that can exist in the ICS, two of them appear as fundamental and may be considered as the "minimal" structures of any ICS. These two relations are the

Figure 2-1: User Interface View of the Database



relations and the attributes relations containing information about all the existing relations and attributes in the database. The attributes of each of these relations may vary and can be defined initially under the principle of "minimality" and later extended as needed.

The **relations** relation should contain at least the name for each of the relations in the database (one tuple per relation). Other attributes may indicate sort order, protection, degree, cardinality, etc.

The **attributes** relation has one tuple for each attribute in the database. The name of the attribute, the name of the relation owning the attribute and the attribute type or domain name is required. If a domain name is chosen, then another relation defining it should also be provided. Other attributes of this relation may indicate default values, constraints, protection, etc.

What information should be added in the form of new attributes to these relations or as new relations in the ICS is the designer's decision.

2.4 The Logical Storage Handler

The storage and retrieval of objects can be identified as a specialised function and therefore left to a Storage Handler component. Given that the internal model is relational, it will be necessary to create, delete and manipulate objects of the model such as relations, attributes and tuples.

It is supposed that the designers use a language such as PS-algol requiring therefore no Physical Storage Handler component. Otherwise, a Physical Storage Handler manipulating objects on buffers and possibly a File Handler communicating with the operating system's I/O functions is necessary. The lowest layer of the example system is, hence, the Logical Storage Handler (LSH). It is in charge of creating and maintaining relational structures as commanded by the Name Handler. The encapsulation of all object manipulation in a dedicated component permits the experimentation with different relational structures to meet an adequate compromise between space usage and processing efficiency.

It may also create, on demand, indexes on relations. The initial functionality required from the LSH is to permit access to individual relations, attributes and tuples. The first functions to be implemented may be those constructing objects such as *create relation*, *create attribute*, *add tuple* and of course, *get tuple*. Destructive operations such as *delete relation*, *delete attribute* and *delete tuple*, etc. can be added on a second iteration.

2.5 The Common Query Evaluator

Given a Name Handler and a Logical Storage Handler, the only component that may be required to start a user interface is a Query Evaluator. It may be first restricted to simple queries and to limited and unchecked update capabilities.

The provision of a common Query Evaluator (QE) for a multi-language system requires the definition of a common Intermediate Query Language (IQL) as discussed earlier. The IQL should be amenable to computer processing and it is therefore not required for it to be particularly user-friendly. With the internal data model being relational, it appears natural to define a relational language, although other alternatives (e.g. FQL [Buneman 82]) may also be tried. In the database environment implemented for this research, the IQL in the form of a relational algebra was defined and implemented in a manner similar to the proposals by Atkinson for an internal query language for the distributed system Proteus [Atkinson 82]. It is briefly presented in the next section.

2.5.1 The Intermediate Query Language (IQL)

The IQL is proposed as a candidate internal database standard. Its form has been chosen to show the following characteristics:

1. A simple subset of it can be implemented easily.
2. It is extensible to meet new processing needs.
3. It is amenable to transformation and therefore an optimizer is feasible.
4. It is machine independent and therefore queries can be transported to any installation implementing it.

The IQL consists of a set of one and zero address instructions intended for a hypothetical stack machine. Each instruction is denoted by a mnemonic word that may also indicate the type of the instruction. Initially the types are one of boolean (b), integer (i), string (s) or relation (r). Other possible types are float (f), time (t), date (d), money (m), etc. Each instruction may be uniquely labelled by a word preceded by a dollar sign. There is a reserved label $start which defines where to start evaluating the query.

As an example, consider a database having a relation "Person" with the attributes "Name", "Sex" and "Address". The query "List the name and address of all females" can be expressed in IQL as in figure 2-2. In most of the instruction mnemonics, it is the first letter

Figure 2-2: Intermediate Query Language: Query 1

```
$start rload Person ! load relation id onto stack
       rselect S    ! select tuples using label S
       rproject P   ! project attributes using label P
       rresult      ! end of main query body

$S     sload "f"    ! load string "f" onto stack
       satt Sex     ! load (string) attribute value
       sequal       ! compare top stack items
       bresult      ! result is a boolean on stack

$P     satt Name    ! load string att. Name on stack
       satt Address ! load string att. Address on stack
       end          ! end of projection list
```

(e.g. r,s,b) that denotes the type of the instruction. The example in figure 2-2 shows the anatomy of an IQL request. Basically, it consists of a main body indicating the relational operations involved in the query, and a label associated with each of them that define how to process the operation.

The text of an IQL query consists of a sequence of statements conventionally one to a line. There are essentially two levels of language, the relational and the scalar language. The relational language defines operations on relations. The scalar language includes operations for each of the scalar types defined.

2.6 The Update Handler

The Update Handler (or integrity subsystem) is in charge of ensuring the database consistency by monitoring update requests to detect integrity violations. It enforces the integrity rules (constraints) defined for the database by taking the appropriate actions if an attempted violation is detected e.g. it may reject the operation, report and record the violation or initiate automatic corrective actions [Eswaran 75]. It may also reject the definition of a new rule if this rule is violated by the current state of the database.

If an update is made on a meta-data relation (e.g. adding a tuple to the "attributes" relation, the Update Handler is in charge of triggering the corresponding procedures as side-effects of the update (e.g. adding the new attribute to the corresponding relation). Triggers are a means of implementing the integrity rules by automatically executing special-purpose procedures whenever a "trigger" event occurs [Date 83]. It is also important that the rules are subject to queries and updates, as any other piece of information in the database.

Date and Hammer and McLeod classify integrity rules in two categories: domain integrity rules and relation integrity rules [Hammer 75, Date 83]. Domain rules describe a domain as a set of atomic data objects and relation rules specify a property or relationship that must hold on or between one or more relations or subparts of relations (tuples or attributes).

Every attribute has an underlying domain which is a subset of one of the *base* domains e.g. the set of all strings, the set of all integers, etc. Domain integrity rules are associated with the definition of the domain which either explicitly or implicitly specifies all acceptable values for the domain. Domain rules are considered in isolation i.e. independent of other attribute values in the database.

In our implementation an attribute may be associated with a predicate that defines its domain. For example, the attribute "age" of a person can be associated with the predicate (age >=0 AND age <= 150). The Update Handler ensures that every new value of this attribute satisfies the predicate. For this purpose, the Update Handler maintains a "compiled" version of the predicate, in the form of a query, that is sent to the common Query Evaluator together with a tuple holding the candidate attribute value. If the stack machine produces a "true" value, the attribute is accepted. Another domain rule allowed is that of enforcing that the values of an attribute in a relation are "unique" i.e no duplicate attribute values may exist.

Relation integrity rules concern the admissibility of a given tuple as a candidate for insertion into a given relation, or the relationship between tuples of one relation and those of another. In our implementation we allow the definition of *referential* integrity rules specifying that the values of a given attribute must also exist as a value in an attribute of another (or the same) relation. For example, the values of the attribute "rel-name" in the ICS "attributes" relation must also exist as values of the "rel-name" attribute in the ICS "relations" relation.

The concentration of all integrity rules in one component, instead of scattered across several components makes these rules easier to maintain, extend and check.

2.7 The Utilities Component

During the implementation of the components, it becomes clear that many housekeeping routines can be shared among components. These routines can be incorporated in a Utilities component and some of them are:

a) I/O (from/to a terminal or an external file): reading a query, reading data and ensuring that it belongs to a given type e.g. reading/checking date or time type input. Displaying help and error messages, querying results, etc. Output to an external file of a relation (in both formatted and unformatted modes). Loading a relation from an external file, etc.

b) Data Type conversion and String handling: our implementation stores all data as strings and therefore requires some type conversion routines: integer to string, time to string, string to integer, etc. as well as some string handling facilities such as extracting the first (next, or any) symbol of a string according to the symbol delimitation rules (e.g. spaces, newlines, tabs).

c) Vector Handling: much of the data (e.g. tuples) is manipulated as string and pointer vectors in our implementation. The following routines facilitate their handling:


```
extend.s.vector (*string vs; string s -> *string)
  given a vector of strings vs and a string s, it
  appends s to vs (extending the vector by one element)
extend.p.vector (*pntr vp; pntr p -> *pntr)
  as extend.s.vector for pointer vectors.
copy.s.vector (*string a -> *string)
copy.p.vector (*pntr p -> *pntr)
shrink.s.vector (*string sv; int i -> *string);
  given a vector of strings sv and an integer i, it
  returns a smaller string vector without the i-th element.
shrink.p.vector (*pntr pv; int i -> *pntr):
  as shrink.s.vector for pointer vectors.
```

d) Functions over names: given a name a component may need to check whether it is a base relation, a query, a view, an attribute name, etc. In the case of an attribute, it may need to know its type, constraints, default value, position, etc. Different routines handle these requests.

2.8 The Optimizer Component

As the volume of data grows and the processing requirements become more complex, a system should include an Optimizer component that establishes the evaluation strategy of a query. The objective is to relieve the users, especially casual users, from the need to know any detail that would make a query yield faster a result. In other words, the speed with which a query is evaluated should not depend critically on the way the query is formulated [Hall 75]. An optimizer will typically calculate the cost of evaluation of a query, due to possibly different evaluating strategies, according to a cost model and select the strategy with minimal cost. The cost may be expressed in both cpu time and number of disk accesses. Jarke and Koch have surveyed many query optimization strategies [Jarke 82].

Yao, Blasgen and Eswaran propose various methods of evaluating a query based on a storage and access model of a relational database [Yao 79, Blasgen 76]. The methods include the use of indexes on join attributes, use of indexes on select attributes, sorting relations and different evaluation orders of the operations involved in the query. A number of cost equations are given that consider various storage and access path parameters. The methods are compared for various parameter values and the conclusion is that they can effectively reduce the time of query evaluation. Blasgen and Eswaran indicate a global strategy for an optimizer: first, given a query, determine the applicable methods from the available access paths. Second, eliminate any obviously bad method and those failing to pass certain simple tests. Finally, evaluate the cost estimates for the remaining methods and choose the method with minimum cost. They conclude that optimization methods should take into account existing access paths and their properties.

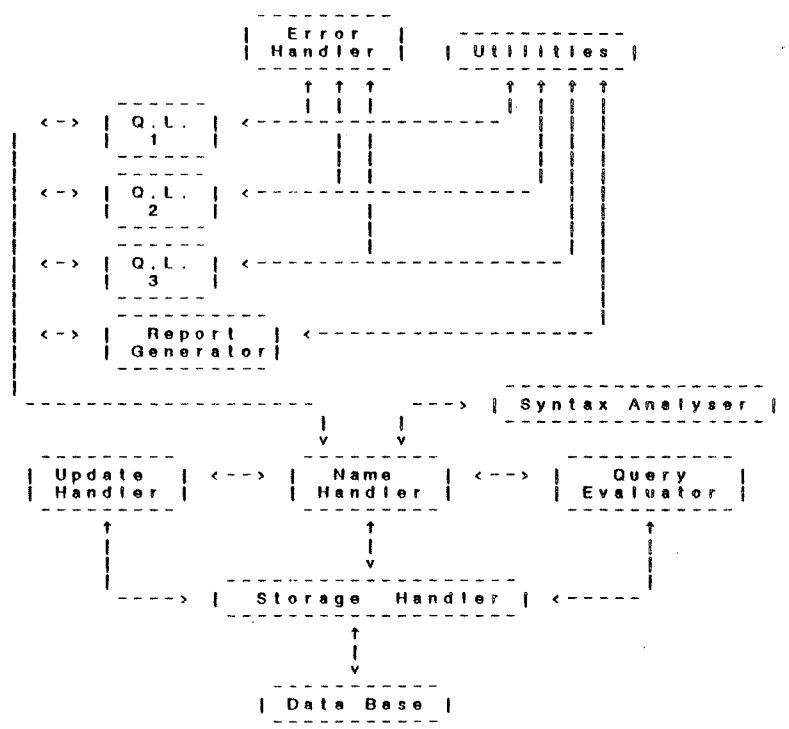
2.9 An Extended Environment

In the implemented environment reported in the next chapter, two internal query languages are used. One is a higher level relational algebra, called RAQUEL (Relational Algebra Query and Update Extensible Language), which is produced by the user interfaces. It is in fact one

of the query languages offered to the users. RAQUEL queries are passed to a common Syntax Analyser component (through the Name Handler) which in turn produces the IQL tree. This second indirection, although introducing some overhead, simplifies the implementation of user interfaces by leaving to the common Syntax Analyser the checking of most of the errors introduced by users when defining a query. It is also considerably easier to generate a query in this relational algebra than in IQL due to the higher procedurality and verbosity of IQL.

The architecture implemented for this research is shown in figure 2-3. To simplify the figure, the use by all of the components of the Utilities component is not shown, nor is the use of the Error Handler by the Name and Update Handlers shown.

Figure 2-3: An Extended Architecture



2.9.1 Name Handler

Before calling other components, relation names are checked for existence and then translated into internal identifiers. If they exist, then the required function is called, otherwise the Error Handler is called. Other parameters are checked at each function. Some functions use or return an object identifier (tuple id or relation id). They are the identifiers of the structure. In the case of tuples, a *tuple id* is a reference to a *tuple header* structure associated with the tuple. If the parameter does not include *id* then a reference to the structure itself is passed. In the case of tuples this is a reference to a vector of strings.

- **open database (database name)**: if a new database is opened, the necessary ICS structures are created (e.g. *relations* and *attributes* relations). It also performs some table initializations to speed up the mapping from names to internal identifiers. At this point, some security mechanisms may be invoked to check the access rights of the user to the database. The parameters of the function may be then extended to include a user identification and password.

- **close database (database name)**: called at the end of a session with a user interface and only if the user decides to *commit* any changes made to the database. It cleans up some temporal tables that may not persist. It is also feasible to collect final session statistics and restructure relations according to these statistics or upon explicit user request.

- **add tuple (relation name, tuple), delete tuple (relation name, tuple), modify tuple (relation name, old tuple, new tuple)**: calls the Update Handler to check protection, possible duplication of tuples and integrity constraints. If the Update Handler judges the operation as valid then the Storage Handler is called to process the operation. These, and other functions listed next, acting presently upon one tuple only, could be easily extended to work on all or a specified number of tuples.

- **find tuple (relation name, target tuple)**: the Storage Handler is called to find a tuple matching all the attribute values of the target tuple. This function may additionally accept a parameter indicating which attribute(s) ought to be consider(ed) in the matching test, and at which tuple to start the test.

- **get tuple (relation name, tuple id, which)**: which is any of first, last, next, or previous. The Storage Handler is called to get the required tuple.

- **store structure (structure name, structure reference), get structure (structure name), drop structure (structure name)**: a user interface may require from the Name Handler to make persistent any structure that may be required in another session. This is used in query languages to store views, snapshots, to leave a message that is recalled automatically in the next session, etc.

- **compile (query name, query string)**: the common Syntax Analyser is called to compile the query and, if correct, translate it into a tree which is returned to the calling interface.

- **evaluate (query name, query tree)**: the object names in the query tree are replaced by internal identifiers before the query is passed to the Query Evaluator.

- cardinality (relation name), degree (relation name), keys (relation name): the Storage Handler maintains the cardinality and degree of each relation in a relation header. The keys of a relation are those attibutes that have the constraint *unique*.

- create index (relation name, attribute name), delete index (relation name, attribute name): indexes are currently limited to one attribute and are implemented using the tables facilities (B-trees [Comer 79]) of PS-algol.

2.9.2 Storage Handler

This component performs the following operations:

- add relation: an empty relation structure is created, returning the new relation identifier.

- delete relation(relation id): the relation structure and associated indexes are deleted.

- add attribute (relation id, attribute type, attribute default: each attribute has associated in its Storage structure a default value and a type that is required by the Query Evaluator.

- delete attribute (relation id, attribute id): deletes the specified attribute.

- add tuple (relation id, tuple), delete tuple (relation id, tuple id), replace tuple (relation id, old tuple id, new tuple): "replace tuple" corresponds to the modify tuple operation of the Name Handler.

- find tuple (relation id, tuple), scan relation (relation id, tuple): the former searches the relation for a tuple matching exactly the target tuple, the latter checks only the non-null attribute values and is used by the Name and Update Handlers only.

- sort relation (relation id, sort spec): this function is used by the Name Handler and Query Evaluator. The sort specification provides the identifiers of the participating attributes, the sort order and whether an ascending or descending order is required.

- build index (relation id, attribute id), delete index (relation id, attribute id): this functions create a new index or destroy an existing one.

- cardinality (relation id), degree (relation id): return the cardinality or degree of the specified relation.

- max width (relation id): this function is used by the report generation facilities of each user interface. It returns a vector of integers where each element indicates the length of the largest attribute value, considering all tuples, for the corresponding attribute (attribute values are stored as strings in this implementation).

2.10 Conclusions

We presented in this chapter a series of components that can be gradually designed, implemented and tested. An important objective is to allow the users of the system to influence the construction of the system throughout its development. This is achieved by providing increasingly complex prototypes that evolve step by step into the final system. By defining suitable levels of functionality for each component, their evolution and that of the user interfaces can proceed in parallel. This was defined as the *minimal functionality* principle which is applied at each implementation stage and serves both to organize the design and implementation activities, providing a yard stick for each stage.

During our experiments in constructing database components we found that the rapid provision of new user interfaces can be achieved by constructing the internal components so that they allow the user interfaces to concentrate on their main tasks of collecting and requesting information to and from the user. This was defined as the *minimum responsibility* principle for user interfaces.

One of the aspects that helps faster building of a database environment is the identification of those tasks that are considered essential, postponing the addition of those less important to a later stage. Some of the traditional database tasks such as concurrency, recovery and security have not been incorporated. The reasons for this are threefold. First, we oriented our research toward personal or small-group databases where these tasks are less likely to become critical. Second, if they do become critical they can be incorporated to the system at a later stage, following the same principles outlined in this chapter. Third, some of them (e.g. concurrency and recovery) may be provided by the underlying software used to build the components (e.g. PS-algol will handle concurrency).

These components, resulting from our implementation experiments, allowed us to construct the database environment described in chapter 3 consisting of different user interfaces. An important finding of these experiments was that the database software can be effectively divided into separate but cooperating activities that can be gradually improved and extended.

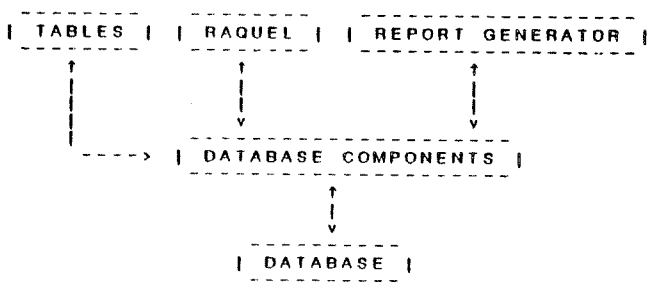
Chapter 3

Implemented Database Environment

3.1 Introduction

This chapter describes part of the implemented database environment, consisting of three user interfaces, built upon the software components dscribed in the previous chapter. All the user interfaces constituting the environment shown in figure 3-1 use the same set of database components, as described in chapter 2 and depicted in figure 2-3.

Figure 3-1: Implemented Database Environment



The user interfaces consist of two relational query and update languages, TABLES and RAQUEL, and a Report Generator. All of them may access the same database with the novel feature that they use an entirely consistent name space. RAQUEL has been used by a number of undergraduate and postgraduate students during the 1982/1983 academic year. The other interfaces have been subject of demonstrations only so far. The whole set of interfaces described in this chapter is expected to be used by students in the coming academic year.

Although many people have used and commented on these interfaces, we regard them, in their present state, as a demonstration of the reusability of the proposed components rather than complete user interfaces. More development is both sensible and necessary.

We have advocated a user oriented approach to database software engineering throughout this thesis. For that reason, we considered that by observing the way query languages are used we could improve their appeal to the users. The EDQUSE results influenced the design of the TABLES and RAQUEL user interfaces in many ways. We had the opportunity to assess this influence during a database exercise set to third year Computer Science students (using a more primitive version of RAQUEL than the one presented in this chapter). Given a choice of RDB (a functional/relational query language [Nikhil 82]), EFDM (a functional query language [Kulkarni 83]), ASTRID (a relational algebra query language [Bell 80]), EDQUSE and RAQUEL, out of 14 groups of three students each, and when TABLES was not available, one chose RDB, one chose EDQUSE and 12 chose RAQUEL to do a variety of database projects. In an earlier database exercise for a group of MSc students, the same languages were analysed by them, deciding finally to use RAQUEL for their project.

The user interfaces are described in general in this chapter, using the database shown in figures 3-2 and 3-3 for examples.

Figure 3-2: University Database. Bachman Diagram

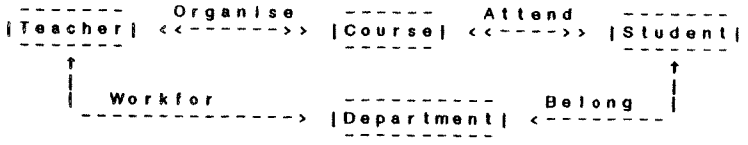


Figure 3-3: University Database. Relational Description

```
Teacher    : t#:integer,name:string,salary:integer
                 phone:integer,ext:integer
Course     : c#:integer,title:string
Student    : s#:integer,name:string,faculty:string
Department : d#:integer,dname:string,head:string
Organise   : t#:integer,c#:integer
Attend     : c#:integer,s#:integer
Workfor    : t#:integer,d#:integer
Belong     : d#:integer,s#:integer
```

TABLES is expected to be used by casual users as well as by those requirying only limited processing capabilities. RAQUEL is a relational algebra that uses a more elaborate mechanism than TABLES to build queries and is fully relational. The Report Generator is an enhanced output facility from the one provided by default by all the query languages. It is also capable of peforming some basic statistics on relations as well as drawing histograms.

In the following presentation of the user interfaces, which is a summary of the manuals, the words *table*, *column* and *row* are sometimes substituted for *relation*, *attribute* and *tuple* respectively.

3.2 TABLES

TABLES is a screen oriented query and update language. The design of the screen handler has drawn heavily from EDQUSE [McDuff 80] which in turn has borrowed many ideas from QBE [Zloof 75]. TABLES is expected to compete with EDQUSE in terms of being easy-to-use.

TABLES models applications as a set of named tables (relations). Each table consists of a set of rows (tuples) and a number of named columns (attributes). All values in a column have the same type which can be one of integer (i), string (s), boolean (b), time (t) or date (d).

Every new database created through TABLES is provided with three tables. These are the *tables*, *columns* and *help* tables. The first two are common to all relational query languages accessing this database and contain information on table names, column names, column types, etc. The latter is particular to TABLES and provides a summary and examples of its use. The *tables* and *columns* tables constitute the TABLES conceptual schema and may be queried and updated as any other database query, provided some associated constraints are satisfied. This is a major difference from EDQUSE which provides no visible schema and provides different mechanisms for adding new tables and columns.

With respect to help facilities (EDQUSE has none), TABLES provides the *help* table plus short summary messages while forming queries, traversing or updating a table.

The following is a layout of the *tables* table after a new database has been created i.e no user created table exist.

tabname	sortorder	protection
s	s	s
columns	tabname=a	dt,ac,dc
help	n	n
tables	tabname=a	dt,ac,dc

The first row is the table header and contains the *column names*. In the above table, these are *tabname*, *sortorder* and *protection*. The next row indicates the column types which in the above case is *string* ("s") for all columns. The subsequent rows contain column values. The fourth row indicates that there is a table whose name is *help* and that it is not sorted and has no protection.

The *columns* table contains the table name, column name, type, default, constraint and position of all columns of all tables in the database. Unlike EDQUSE, TABLES allows the user to define integrity constraints associated with column values, protection to tables, columns and rows, default values and sort order. The meaning and use of each of these columns is explained in the presentation of RAQUEL.

In addition to the above structures, a user may define *views* which are named queries that persist between TABLES sessions. The view mechanism is explained later. This facility does not exist in EDQUSE and may significantly enhance the usability of the interface for users defining large and repetitive queries that may persist as "views".

3.2.1 TABLES Queries

Like QBE but unlike EDQUSE, TABLES queries have a name and can be used in building further queries. This is an important advantage over EDQUSE in which a complicated query cannot be broken into smaller queries. Furthermore, we have observed that understanding and managing queries becomes difficult if the query involves more tables than the screen can hold at one time (5 tables at most). With TABLES, a query can be systematically built out of simpler queries involving a screen full of tables each time.

A query is formulated by placing elements on a skeleton table displayed on the screen. Three relational operations are implemented: project, select and join (these are explained in some detail in section 3.3.1). Projecting and selecting are achieved in the same way as in EDQUSE or QBE.

In order to link information from two tables, EDQUSE and QBE use "example" elements that are inserted in the columns participating in the join. The analysis of EDQUSE query errors indicates a high rate of errors in queries on more than one table. This is mainly due to the incorrect use of example elements. In an attempt to improve that situation, TABLES uses a different approach. The user does not insert example elements but indicates the tables and columns participating in the join. TABLES checks that the columns have a matching type and inserts, between brackets, in each column participating in the join, the name of the other table.

The following example (figures 3-4 and 3-5) illustrates the dialogue between the user and TABLES to join the tables "Teacher" and "Organise" through the columns "t#" (which have the same name but this need not to be so).

The user first asks TABLES to display the table "Teacher" (Command: get Teacher). TABLES displays a row with column names, followed by a row with column types, and an "empty" row in which the user can insert elements. A cursor, shown in figure 3-4 as "#" and placed initially on the first column, is used to traverse the table.

Figure 3-4: TABLES Display of a Skeleton Table

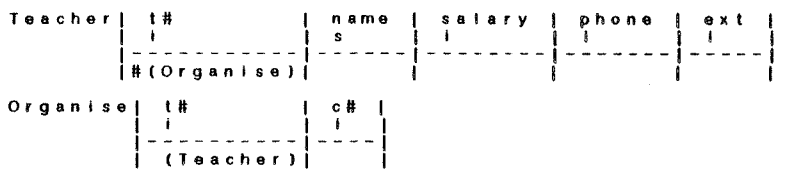
Teacher	t#	name	salary	phone	ext
	i	s	i	i	i
	#				

The cursor can be moved to other columns or rows but must be positioned at column "t#" when starting a join to indicate that "t#" is the column of the "Teacher" table participating in the join.

The user then indicates its intention to join the "Teacher" and "Organise" tables (Command: join). TABLES will ask for a table name ("Organise") which is displayed. It will then ask for the join column name of Organise ("t#") and check that it exists and both

columns have the same type ("s"). If the join is valid, TABLES will insert the table names in the corresponding columns as in figure 3-5.

Figure 3-5: TABLES Join Query



The join operation is thus closely guided and checked by TABLES and it is explicitly indicated in the skeleton tables. This new approach has yet to be tested with users to assess its effectiveness. An improvement to this method would be to insert not only the join table name but also the join column name. Self-joins, or joins on more than one column would then be more properly indicated.

3.2.2 TABLES Commands

A TABLES session starts in *command mode* from which a number of commands can be used, including some that will enter another mode, from which the user may return by providing a "null" command (press RETURN key in response to the command prompt).

To display a table on the screen, the user types the name of the table which is then displayed without any interposed evaluation. With EDQUSE, the user must define a query over the table (possibly projecting all columns), which is then analysed, evaluated and displayed. The simpler approach used in TABLES may save a considerable amount of time since during a session, existing tables or query results may be re-displayed many times. This limitation of EDQUSE may account for the large proportion of "project only" queries recorded.

Once a table has been displayed, the user enters *traverse mode*. A cursor is positioned on the first row, first column of the table. The following commands describe those available in TABLES when in the traverse mode.

- bottom : move cursor to the last row of the table displaying new rows if necessary.
- cfind : (find value in column) search the table for a value in the current column that matches the given value. Position the cursor in that row. If no value is found, leave the cursor at the bottom row.
- down : move cursor to the next row, if any.

- find : as cfind but search in all columns.
- help : display names of commands that can be used in the traverse mode.
- left : move cursor to the column at the left, if any.
- right : move cursor to the column at the right, if any.
- top : move cursor to the first row in the table, re-displaying rows if necessary.
- up : move cursor one row up, if any.

The following list describes the commands available in TABLES when in command mode:

- exit : terminate TABLES session. The user decides whether to *commit* or not. If the answer is positive, any updates performed during the current session will be applied. Otherwise the database preserves the state it had before the start of the session.
- drop : a given "view" name is dropped from the views table.
- help : display the *help* table.
- include : a given query name is included as a view.
- load : bulk load of rows from an external file.
- queries : display the query names and the names of the tables included in each query.
- query: define a new query or edit an existing one. A query name must be given. If a query with that name exists, then some of the tables it includes (up to a screen full) are displayed as skeleton tables on the screen. The user may then use the following commands (listed in alphabetical order), to complete the specification of a query:
  - constant : define a constant element in the current column. The result query must have in this column, values that match the constant element. (This implements the "select" operation).
  - down : move the cursor to the next row down. Create a new "empty" row if necessary.
  - forget : the user is asked for a table name that will be excluded from this query. A query may not have any tables isolated i.e. not joined to the other tables. To avoid that syntactic error, a table which has been obtained by get must be excluded by forget if it is not needed.
  - get : the user is asked for a table name whose skeleton will be displayed on the screen (if it is not already there) and the cursor positioned on its first column.
  - help : display message with available commands for queries.

- o join : used to join tables, as described in the example in section 3.2.1.
- o kill : delete the element in front of the cursor, if any.
- o left : move the cursor to the column at the left, if any.
- o next : move the cursor in front of the next element in the current column, if any.
- o print : insert a "p." element in the current column to indicate that this column must be included in the result table.
- o right : move the cursor to the column at the right, if any.
- o tables : list the names of the tables participating in this query.
- o up : move the cursor one row up. Stop before entering the row with column types.
- o =, >, <, <=, >=, <> : similar to constant (which defines equality), these commands define the relationship that must exist between the given element and the column values in this table, in the result table.
- o A "blank" response to the command prompt will terminate the definition or edition of this query and return to command level.

- output : produce a formatted output file from the given table or query name.
- update : display a given table name and enter update mode. This mode allows the use of all the commands defined for the traverse mode plus the insertion and deletion of rows and the modification of column values. Any update is applied immediately i.e. what the user sees on the screen is what is actually in the database. The user is not required to run a query on a table before updating it (as in EDQUSE) but the named table is directly displayed. The most important difference from EDQUSE is that in TABLES the update mechanism applies to any table in the database, including those constituting the schema (the tables and columns tables). A new table is added to the database by adding a row to the *tables* table. A similar procedure is used to add new columns.
- views : display current views and the names of the tables involved in each view.

3.3 RAQUEL

RAQUEL is an interactive program with facilities for querying, updating and manipulating a database. It is a relational algebra and its design has been influenced mainly by Astrid [Bell 80]. TABLES users may regard RAQUEL as the natural next step in terms of processing power since both RAQUEL and TABLES can access the same database and both use a consistent name space. Compared to TABLES, RAQUEL is more suitable for expert users, although it has been successfully used by non-experts, as indicated in the introduction. Its main advantage over TABLES relies on its greater processing capabilities. It is fully relational and has grouping functions. It has uniform access to both data and meta-data.

As in TABLES, a database is organised as a collection of named tables, three of which are provided with every new database. these are the *tables*, *columns* and *helpraquel* tables.

The *helpraquel* table contains information on the commands and operations that can be performed in RAQUEL.

Data Types and Type Operations: the data types supported by RAQUEL are: integer (i), string (s), boolean (b), time (t) and date (d).

The following relational operators may be used:

= : equal to
< : less than
<= : less than or equal to
> : greater than
>= : greater than or equal to
<> : not equal to

The order for strings is lexicographic, for booleans is "F" < "T".

Integer operations:

+, -, * addition, subtraction, multiplication,
/, rem division, remainder.
- sign inversion.

String operations:

in: Substring e.g. "bc" *in* "abcde" yields true

++: Concatenation e.g. "abc" ++ "de" yields "abcde"

Boolean operations:

and, or, not

Notation: a *name* is an arbitrary length sequence of letters , digits and the characters '#' and "_".

The syntax of the operations on tables is explained through examples and also using the following more formal notation:

- all symbols are introduced as expressions where the left-hand side introduces a name for a symbol and the right-hand side defines the symbol.
- non terminal symbols (defined in terms of other symbols) are specified as identifiers and terminals as string constants (between apostrophes).
- names to be defined by the user are stated between angle brackets.
- alternatives are separated by a vertical bar.
- a right-hand side expression may contain braces to delimit expressions which can occur zero or more times.

Example. positive numbers can be represented as:

```
number ←  digit | number digit
digit  ←    '0' | '1' | '2' | '3' | '4' |
            '5' | '6' | '7' | '8' | '9'

alternatively a number may be defined as:

number ← digit { digit }
```

3.3.1 RAQUEL Queries

A query is defined through the command "query". Queries are terminated by a semicolon and may be spread over several lines.

A query produces a temporary table i.e. it is not included in the *tables* table and cannot be updated. Note that since a query defines a table, previous queries may be used when defining further queries. This is very useful in building complex queries in terms of simpler ones.

The operations are of type *unary*, operating over one table or *binary* operating over two tables. One query definition may invoke many operations.

The flavour of RAQUEL will be presented in the following examples. "C:" is the RAQUEL command prompt and the text at its right is entered by the user. If the text starts with the character "!" then it is a comment and is not processed. A table (or a query result) is displayed by typing its name in response to the command prompt.

Unary Operators.

The Project operator forms a vertical subset of an existing table. Redundant duplicate rows in the result table are removed. An example is shown in figure 3-6.

Figure 3-6: Raquel: Project Operation Example

```
C: ! project the tabname and colname columns
C: ! from the tables table producing a result
C: ! table named tabcol:
C: query tabcol := tables projected to tabname,sortorder;
C: tabcol

tabcol:
 ------------------------
| tabname     | colname  |
| s           | s        |
|-------------|----------|
| columns     | tabname=a|
| helpraquel  | n        |
| tables      | tabname=a|
 ------------------------
```

```
The   syntax is:

project  ← <table-name> proj-op col-list
proj-op  ← any abbrev. of word 'projected'.
           The word 'to' is optional
```



```
col-list ← <column-name> {',' <column-name> }
```

The Select operation forms a new table by taking a horizontal subset of an existing table, that is, all rows of an existing table that satisfy a given condition. An example is shown in figure 3-7.

Figure 3-7: Raquel: Select Operation Example

```
C: ! Produce a table named onlytab that contains
C: ! information about all columns of the "tables" table.
C: query onlytab := columns selected on tabname = "tables";
C: onlytab

onlytab:
 ------------------------------------------------------
| tabname | colname     | type | default | constraint |
| s       | s           | s    | s       | s          |
|---------|-------------|------|---------|------------|
| tables  | tabname     | s    |         | unique     |
| tables  | sortorder   | s    |         | n          |
| tables  | prtection   | s    |         | n          |
 ------------------------------------------------------
```

Note that in the formulation of the query, "tables" is a column value (not a column name as tabname) of type string and therefore must be presented between quotes. Parenthesis may be used to clarify evaluation order.

```
The syntax is:
select       ← <table-name> select-op expression
select-op    ← any abbreviation of the words
               'selected', 'where' or 'with'.
               The word 'on' is optional
expression   ← or-exp 'or' and-exp | and-exp
and-exp      ← and-exp 'and' ineq-exp | ineq-exp
ineq-exp     ← add-exp ineq-op add-exp | add-exp
ineq-op      ← '<' | '<=' | '>' | '>=' | '=' |
               '<>'| 'in'
add-exp      ← add-exp add-op mult-exp | mult-exp
add-op       ← '+' | '-' | '++'
mult-exp     ← mult-exp mult-op factor | factor
mult-op      ← '*' | '/' | 'rem'
factor       ← primary | sign primary
sign         ← '+' | '-' | 'not'
primary      ← atomic | '(' exp ')'
atomic       ← const | ident
const        ← integer-const | bool-const |
               string-const
integer-const ← digit {digit}
digit        ← '0' | '1' | ... | '9'
bool-const   ← "T" | "F"
string-const ← '"' string '"'
string       ← char {char}
char         ← 'a' | ... | 'z' | 'A' | .. | 'Z' |
               '#' | '_'
```

The Modify operator is used to modify column values in rows for which a given predicate is true. The expression and the column must have the same type. It is normally used in update operations to modify an existing table but it may also used in queries, creating a result table, as shown in the example of figure 3-8.

```
The syntax is:
```

Figure 3-8: Raquel: Modify Operation Example

```
C: ! Create a table modifying the extension number to "77"
C: ! for those teachers having extension number = "0"
C: query new_ext := Teacher modified on
                    if ext = 0 then ext := 77;
C: ! show the Teacher and the new_ext tables
C: Teacher

Teacher:
 ------------------------------------------
| t#  | name        | salary | phone | ext  |
| i   | s           | i      | i     | i    |
|-----|-------------|--------|-------|------|
| 1   | Alicia      | 35000  | 24    |    0 |
| 2   | Margarita   | 24000  | 23    |   21 |
| 3   | Adriana     | 18000  | 22    |    0 |
 ...   .......       .....    ..        ..

C: new_ext

new_ext:
 ------------------------------------------
| t#  | name        | salary | phone | ext  |
| i   | s           | i      | i     | i    |
|-----|-------------|--------|-------|------|
| 1   | Alicia      | 35000  | 24    |   77 |
| 2   | Margarita   | 24000  | 23    |   21 |
| 3   | Adriana     | 18000  | 22    |   77 |
 ...   .......       .....    ..        ..
```

```
modify    ← <table-name> modif-op if-exp
modif-op  ← any abbrev. of words 'modify'
              or 'modified'. Word 'on' is optional
if-exp    ← 'if' expression 'then' assign
assign    ← <column-name> ':=' expression

see select operation for definition of expression.
```

The Order operator is normally the last in a query definition. It produces a result table ordered by column values in either ascending or descending order. An example is shown in figure 3-9.

Figure 3-9: Raquel: Order Operation Example

```
C: ! produce a table with the teacher's name in ascending order
C: query neat := Teacher ordered on name = a;
C: neat

neat:
 ------------------------------------------
| t#  | name        | salary | phone | ext  |
| i   | s           | i      | i     | i    |
|-----|-------------|--------|-------|------|
| 3   | Adriana     | 18000  | 22    |    0 |
| 1   | Alicia      | 35000  | 24    |    0 |
| 2   | Margarita   | 24000  | 23    |   21 |
 ...   .......       .....    ..        ..
```

Syntax:


```
order     ← <table-name> order-op ord-spec
ord-spec← <column-name> '=' ord
            {, <column-name> '=' ord}
order-op← any abbrev. of word 'ordered'.
            The word 'on' is optional.
ord       ← 'a' | 'd'
```

The Extend operator is used to extend a table by appending a new column whose values are determined by an expression. The <col-list> indicates which columns from the argument table must be included in the result table (as in the project operator).

The expression can be formed using the same operators as with the select operator. The type of the expression determines the type of the new column. An example is shown in figure 3-10.

```
Syntax:
extend    ← <table-name> ext-op  ext-exp
ext-op    ← any abbrev. of word 'extended'
             the word 'to' is optional
ext-exp ← col-list ':' <new-col-name> ':=' exp
col-list← <column-name> {',' <column-name> }

See select operation for definition of 'exp'
```

Figure 3-10: Raquel: Extend Operation Example

```
C: ! produce a table similar to "Teacher" with
C: ! name and salary of each teacher and a new
C: ! column showing its salary increased by 1 %
C: query new_salary := Teacher extended to
     name, salary : new := salary + (salary / 100);
C: new_salary

new_salary:
 ------------------------------
| name        | salary | new   |
| s           | i      | i     |
|-------------|--------|-------|
| Alicia      | 35000  | 35350 |
| Margarita   | 24000  | 24240 |
| Adriana     | 18000  | 18180 |
 .......       .....    .....
```

The example in figure 3-10 shows that the extend operator is a combination of the project and modify operators, but instead of modifying the values in an existing column, a new column is produced. As in all other operations, a query does not modify the values of the tables involved in the query.

The Group operator [Gray 82] is used to partition the argument table into row groups such that within each group all rows have the same value in the column(s) indicated in the col-list and then to compute an expression over each group of rows. The expression should contain aggregate functions from the following list:

group function	expression type
count	-
sum (expression)	integer

```
max  (expression)        integer
avg  (expression)        integer
min  (expression)        integer
all  (expression)        boolean
any  (expression)        boolean
```

all and any are useful in forming quantified queries. all computes the boolean conjunction for each group (the result is true if all members in the group are true) and any computes their disjunction (which is true if any is true).

```
Syntax:
group     ← <table-name> group-op group-exp
group-op  ← any abbrev. of word 'group'.
            The word 'on' is optional.
group-exp← col-list ':' <new-col-name> ':=' expr
col-list ← <column-name> {,<column-name>}

expr : expression as in select operation.
```

The col-list indicates the columns to be considered to form the row groups. <new-col-name> is the name of the new column which will contain the value of the group expression. An example is shown in figure 3-11.

Figure 3-11: Raquel: Group Operation Example

```
C: ! Count the number of students in each faculty
C: query students_in_faculty := Student grouped on
       faculty : total := count;
C: ! display "Student" and "students_in_faculty" tables
C: Student

Student:
 ----------------------------
| s#  | name         | faculty |
| i   | s            | s       |
|-----|--------------|---------|
| 5   | Leonor       | SCIENCE |
| 6   | Magdalena    | ARTS    |
| 7   | Veronica     | ARTS    |
| 8   | Andrea       | SCIENCE |
| 9   | Alejandra    | SCIENCE |
 ----------------------------

C: students_in_faculty

students_in_faculty:
 ------------------
| faculty | total  |
| s       | i      |
|---------|--------|
| ARTS    | 2      |
| SCIENCE | 3      |
 ------------------
```

In order to make it easier to update the meta-data, a notation for a table constant was introduced. It can be seen as a one-row table, with no name, and with the column types defined by the column values. Examples are shown in the update operations of section 3.3.3.

```
Syntax:
  const-exp ← '[' column-value-list ']'
```


Binary Operators.

The Join operation. If two tables have a common column type then they may be joined over those columns. The result of the join operation is a new table in which each row is formed by joining together two rows, one from each of the argument tables, such that the two rows concerned have the same values in the common column.

The join column from the right-hand side argument table is not included in the result table. Duplication of column names in this table are avoided by prefixing with "#" the names of the columns from the right-hand side table that also exist in the left-hand side table. An alternative procedure, not yet implemented, is to force a renaming before the join if the tables have common column names other than those participating in the join. An example is shown in figure 3-12.

Figure 3-12: Raquel: Join Operation Example

```
C: ! join the tables column and tables by their common
C: ! column tabname.
C: query tab_col := tables joined through tabname = tabname columns;
C: tab_col
(only part of the result table (tab_col) is shown)

 ------------------------------------------------------------
| tabname    | sortorder  | protection  | colname ...
| s          | s          | s           | s
|------------|------------|-------------|------------
| columns    | tabname=a  | dr,aa,da    | position
| columns    | tabname=a  | dr,aa,da    | constraint
  ......
```

```
Syntax:

join ← <table-name > j-op j-exp <table-name>
j-op ← any abbreviation of the word 'joined'.
j-exp← <column-1> '=' <column-2>
```

The Outer Join operation is similar to *join* with the difference that for those rows in the left-hand side argument table which do not have a matching row in the right-hand side table, a *dummy* right-hand side row is created with default values. An example is shown in figure 3-13. The above example illustrates several aspects of the outerjoin operation. First, the result table has at least the cardinality (in this example the same) of the left-hand side table x. Non-matching rows from this table are included in the result table z by appending default values ('0' in our example) in the columns of the other table (e.g. {1,5,0} and {3,7,0}). Second, the addition of non-matching rows applies to the left-hand side table only: it is a unidirectional outer join (note that row {5,8} of table y does not appear in the result table z). Finally, tables x and y have identical column names. To avoid duplicate names in the result table z, this table maintains the column names of table x, eliminating the join column of table y and renaming the other column b to #b.

```
Syntax:
```

Figure 3-13: Raquel: Outerjoin Operation Example

```
C: ! Consider the table z resulting from
C: ! outerjoining the tables x and y.
C: query z := x oj a=a y
C: x

x :   ---------
      | a | b |
      | i | i |
      |---|---|
      | 1 | 5 |
      | 2 | 6 |
      | 3 | 7 |
       ---------

y :   ---------
      | a | b |
      | i | i |
      |---|---|
      | 2 | 9 |
      | 5 | 8 |
       ---------

z :   -------------
      | a | b | #b |
      |---|---|----|
      | 1 | 5 |  0 |
      | 2 | 6 |  9 |
      | 3 | 7 |  0 |
       -------------
```

```
o-join← <table-name> o-j-op j-exp <table-name>
o-j-op← 'oj' | any abbrev. of word 'outerjoined'

see join operation for the definition of j-exp
```

The **Set Union, Set Difference and Set Intersection** operations are the normal set operations. The tables must have the same number of columns (same degree) and the type of a column in one table must be identical to the type of the corresponding column in the other table. That is, the i-th column in both tables must be of the same type.

The column names of the result table are inherited from the column names of the left-hand side table.

```
Syntax:
set-exp← <table-name> set-op <table-name>
set-op ← '+' | any abbrev. of word 'union'
         '-' | any abbrev. of word 'difference'
         '.' | any abbrev. of word 'intersection'
```

An example, using the same x and y tables of figure 3-13 is shown in figure 3-14.

Figure 3-14: Raquel: Set Operations Example

```
C: query setunion := x + y
C: query setdiffe := x - y
C: query setinter := x . y
```

3.3.2 RAQUEL Commands

As in TABLES, RAQUEL provides a view mechanism. Views are used to provide a different appreciation of the database than the one provided by default by RAQUEL. It can also be used as an alternative naming system and as a means for maintaining the definition of frequently used queries.

Consider a query that is often used throughout sessions. To avoid redefining it each session, the query may be stored as a *view*, which will make it readily available in further sessions.

The following list describes the RAQUEL commands:

- *include* <qry> : include a given query as a permanent view.
- *drop* <view> : delete an existing view.
- *views* : display current views.
- *query* : used for defining a query
- *update* : used for defining an update
- *load* <file> : add rows from external file <file> to an existing table.
- *dump* <file> : dump content of a table into external file <file>
- *output* <file> : formatted output of a table to file <file>.
- <table-name> : display a table on the terminal.
- *queries* : display queries and their definition.
- *exit* : terminate session. The user is asked whether to commit or not. If the answer is positive, any changes made to the database during the session will be applied.
- *keepr* <table name> : used to save a result table as a snapshot. The table is made persistent, allowing the Report Generator to use it.
- *dropr* <table name> : the persistence of this table is not required any more. This command is related to *keepr*.

- *!* : a command line starting with a "!" character is not processed.

3.3.3 RAQUEL Updates

Adding Rows

```
Syntax:
update  ← <table-name> ':=' union-op  table-exp
union-op← '+' | any abbreviation of word 'union'
```

table-exp is any expression using table operations.

Example: add tables 'x' and 'y' to the database.

```
C: update tables := union {"x","n","n"};
C: update tables := union {"y","n","n"};
```

In the above example the table constant notation {..} was used as a table expression. In the same form, new columns may be added to a table by updating the columns table. Example: add two integer columns named 'a' and 'b' , with default '25' and '-1', to the 'x' table:

```
C: update columns := union {"x","a","i","25",""};
C: update columns := union {"x","b","i","-1",""};
```

Deleting Rows is symmetric to adding rows. The operation used is set difference.

```
Syntax:
delete ← <table-name> ':=' diff-op table-exp
diff-op← '-' | any abbrev. of word "difference"
```

Modifying Rows The syntax is identical as for the query case. The difference is that the modification will affect a (permanent) table. Example: assume that a table 'x' has in column 'a' values that are < 0. Modify them to be equal to the sum of the values in columns 'b' and 'c'.

```
C: update x := modified on if a < 0 then a := b+c;
```

Note that the modify operation is similar to the extend operation, the difference being that extend creates a new column and modify modifies an existing one.

3.3.4 Protection, Constraints and Defaults

PROTECTION is defined at the table level (in the tables table). By default, a table is unprotected (protection="n"). The protection may be defined when creating a new table i.e. adding a row to the tables table or by modifying a row in the tables table. A table can be protected against the following events:

```
dt : delete table
ac : add column
dc : delete column
at : add row
dr : delete row
mr : modify row
```


At the column level, a column can be defined as having one of three possible *CONSTRAINTS*. RAQUEL will check that during a loading or update operation, the constraints are not violated. If they are, the offending row will be neither updated nor loaded but the operation will continue.

- *"unique"*: the column value cannot be duplicated in the same column in any other row in the table.
- *!sin* <table-name>.<column.name> : Referential integrity constraint. It specifies that the column values in this column must exist in some row, in the given column of the given table (see the *columns* table for examples).
- *<Predicate>* : any boolean expression that must yield true when adding a new value.

Example of predicate constraint when adding a column to the *staff* table:

```
C: ! Add an 'age' column to the "teacher" table,
C: ! where  0 <= age <= 150
C: update columns := u {"Teacher","age","i","25",
            "(age>=0) and (age<=150)"};
```

DEFAULT values for columns are defined when creating them. All columns must have a default value which will be applied, if necessary, when updating a table (e.g. adding a row without knowing all column values needed). In the previous example, the column age was given the default value '25'. All existing rows in the table are automatically extended to include this column value. If a default value is not supplied when defining a column, RAQUEL provides one for each type: '0' for integers, the null string "" for strings, 'false' for booleans and the current date and time for these types.

3.4 Report Generator

The Report Generator is an interactive program that can access the same databases used by RAQUEL and TABLES. In addition to report generation facilities, this program also provides some basic statistic capabilities. The user specifies which database table is to be used for the report or the statistic analysis. Various commands, described below, may be used to shape the report by modifying a set of default parameters. The effect of each of the commands can be observed on the screen. This also applies to the statistic analysis.

Since the report and statistics facilities ought to be used by all query languages, we decided to combine them in a single user interface that is described below. It is expected that its functionality will be gradually extended and finally derive into two user interfaces. One would be dedicated to report generation and the other to statistics and graphs. What is described next is only a small step towards that aim.

The Report Generator responds to the following commands:

- tables : display the database tables names i.e those in the tables table. Other tables may exists in the database that can be used for report generation e.g. those stored using the command keepr in RAQUEL.

- table <table name> : defines a table for report generation or statistic analysis. All further commands will apply to this table, until another table command is issued.

- title : defines the report title. The user can provide a set of textlines that will be inserted, alone, on the front page of the report.

- rtline <line number> : replaces report title line in current report title.

- ptitle : defines a page title. Every page of the report may begin with a set of textlines as defined by the user. All pages, except the front page will carry the same page title.

- rpline <line number> : replaces page title line in the current page title.

- header : defines page headers. Every page can be associated with a different page header that is printed before the page title. It consists of one line of text.

- rhline <line number> : replaces a page header line.

- chonly <column number> : print column value only when it changes, further identical values are replaced by spaces.

- chname <column number> : change specified column name in current table.

- chpage <column number> : start a new page each time the column value changes.

- display : display current table (report) on the screen. The table is formatted by default, i.e. all columns have the same length and values are justified to the left.

- plength <page length> : define page length in terms of number of rows in the table. Each page number will be indicated in the report.

- schar <new separator character> : change the current separator character. Column values in the report are separated by spaces and a separator character, which is by default equal to "|" but may be changed on request.

- uchar <new underline character> : changes the current underline character. The names of the columns (inserted on each new page) are separated from the table rows by an "underline" line. This underlining is also performed after the last row of the table. The character for underlining is "-" by default but may be changed on request.

- status : display status of report (table name, underline character, page length, etc.).

- help : display the name and a short explanation of each of the available commands.

- exit : finishes report generation session.

The statistic facilities work on a sample constituted by all the values of a column in the current table. These facilities are:

- basic <column number> : generates "basic" statistics using the specified column (which must be of type integer) of the current table. The "basic" statistics are: average, variance, range of sample, maximum, minimum and typical value. Number of occurrences and percentage of sample are given where relevant (e.g. the maximum value in the sample, the number of times it occurred and the percentage of occurrence with respect to the size of the sample). These statistics are given for the whole sample and for the sample without the maximum and minimum values.

- histo <column number> : generates a histogram from the values of the specified column.

- graph : generates an output file for a graphic program. The graphic devices in the department can by used through dedicated programs that require data and parameters (e.g. graph title, units, etc.) in a particular format. This command produces a file from the current table, with such a format.

3.5 Conclusions

The software described above was designed, implemented and tested incrementally. After RAQUEL and the underlying components were implemented, TABLES and the Report Generator were produced in less than one man/week each. We were able to achieve this because we could concentrate on the user interface. All the underlying database software was provided by the existing components used to implement the query languages.

All of the user interfaces were implemented using (re-using) some or all of the software components described in chapter 2. All access the same data, using the same name space. They provide different levels of functionality and usability to suit different needs and preferences. All the interfaces described in this chapter provide ample room for improvements and extensions and the underlying components have proved to offer an effective basis for providing rapidly and economically new interfaces.

Our experience has shown that there are two principal factors that contribute to the rapid prototyping of database systems. One is the provision of appropriate implementation tools, particularly a programming language such as PS-algol with persistence capabilities. The other factor is the availability of adequate database software components such as those proposed in chapter 2. We believe that the right choice of both significantly reduces costs and time in producing database tools, as demonstrated in this chapter.

References

[Atkinson 82] Atkinson, M.P. Gray, P.D.M and Hepp, P.E.
Proteus Working Paper E2.
Technical Report, University of Edinburgh, Computer Science Department, 1982.

[Atkinson 83] Atkinson, M.P et. al.
An Approach to Persistent Programming.
1983.
To be published in The Computer Journal.

[Bell 80] Bell, R.M.A.
Automatic Generation of Programs for Retrieving Information from Codasyl Data Bases.
PhD thesis, University of Aberdeen, 1980.

[Blasgen 76] Blasgen, M.W. and Eswaran, K.P.
A Comparison Of Four Methods For The Evaluation Of Queries In A Relational Data Base System.
Technical Report, IBM Research Laboratory, San Jose, California, February, 1976.
RJ 1726 (25344).

[Buneman 82] Buneman, P. Frankel, R.E. and Nikhil, R.
An Implementation Technique for Database Query Languages.
ACM Transactions on Database Systems 7(2):164-186, June, 1982.

[Codd 79] Codd, E.F.
Extending the Database Relational Model to Capture More Meaning.
ACM Transactions on Database Systems 4(4):397-434, December, 1979.

[Comer 79] Comer, D.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Date 83] Date, C.J.
The Systems Programming Series. Volume 2: *An Introduction to Database Systems.*
Addison-Wesley, 1983.

[Eswaran 75] Eswaran, K.P. and Chamberlin, D.D.
Functional Specifications of a Subsystem for Data Base Integrity.
In *1st International Conference on Very Large Data Bases.* September, 1975.

[Gray 82] Gray, P.M.D.
The Group-By operation in Relational Algebra.
Technical Report, University of Aberdeen. Dept. of Computer Science, 1982.

[Hall 75] Hall, P.A.V.
Optimisation of a Single Relational Expression in a Relational Data Base System.
Technical Report, IBM UK Scientific Centre, June, 1975.
UKSC 0076.

[Hammer 75] Hammer, M.M. and McLeod, D.J.
Semantic Integrity in a Relational Data Base System.
In *1st International Conference on Very Large Data Bases.* September, 1975.

[Hepp 83] Hepp, P.E.
A Database System Architecture Supporting Coexisting Query Languages and Data Models.
PhD thesis, Department of Computer Science, University of Edinburgh, 1983.

[Jarke 82] Jarke, M. and Koch, J.
A Survey of Query Optimization in Centralized Database Systems.
Technical Report, Center for Research on Information Systems, New York University, November, 1982.
CRIS 44, GBA 82-73 (CR).

[Kulkarni 83] Kulkarni, K.G.
Extended Functional Data Model - User Manual.
Technical Report, University Of Edinburgh. Computer Science Department, 1983.

[McDuff 80] McDuff, A.D.
Implementation of Query-By-Example.
Technical Report, University of Edinburgh, Computer Science Department, May, 1980.
CS4 Project Report.

[Nikhil 82] Nikhil, R.
RDB - A Relational Database Management System.
Technical Report, Department of Computer Science and Information Science, University oh Pennsylvania, January, 1982.
User Manual.

[Shipman 81] Shipman, D.W.
The Functional Data Model and The Data Language Daplex.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[Yao 79] Yao, S.B.
Optimization of Query Evaluation Algorithms.
ACM Transactions on Database Systems 4(2):133-157, June, 1979.

[Zloof 75] Zloof, M.M.
Query By Example.
In *National Computer Conference*, pages 431-438. AFIPS, 1975.