NEW FEATURES FOR A RELATIONAL DATABASE SYSTEM TO SUPPORT COMPUTER AIDED DESIGN

by

A. Guttman

Memorandum No. UCB/ERL M84/52

12 June 1984

COVER PAGE

NEW FEATURES FOR A RELATIONAL DATABASE SYSTEM TO SUPPORT COMPUTER AIDED DESIGN

by

A. Guttman

Memorandum No. UCB/ERL M84/52

12 June 1984

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

New Features for a Relational Database System to Support Computer Aided Design

by

Antonin Guttman

New Features for a Relational Database System
to Support Computer Aided Design

Ph.D.

Antonin Guttman

Computer Science Division
Department of EECS

Sponsor:
National Science Foundation

Michael Stonebraker
Committee Chairman

ABSTRACT

Data management is a significant problem in Computer Aided Design (CAD) systems which handle large volumes of data in a variety of forms. However traditional database systems have not been widely used in CAD because they are hard to use and do not offer some needed functions. The purpose of this research is to develop features that can be added to relational database systems to make them more useful for supporting CAD.

We ran an experiment in which a prototype relational database system, INGRES, was used to store Very Large Scale Integrated circuit (VLSI) design data, and to perform typical CAD retrieval operations. INGRES was awkward to use in this context and ran much slower than specially designed CAD programs. The remainder of this thesis investigates several additions to INGRES suggested by the experiment that would make it easier to use and more efficient.

We augmented the relational query language QUEL with an *append** command for expressing transitive closure queries, which are needed to retrieve hierarchically

structured VLSI designs. Processing issues are discussed and implementation test results are given.

An indexing structure called an R-tree is presented which can be used to retrieve data for overlapping non-point spatial objects according to their locations in a multi-dimensional space. R-trees are designed for data residing on paged secondary memory, and are suitable for use as a database access method.

Specialized applications like CAD often assign meaning to data that cannot be exploited in a general-purpose database system to optimize data structures and query processing. We outline a way for user-supplied semantic routines to supply information to the database system for this purpose.

CAD applications often use composite design entities made up of many parts. We discuss three ways to represent such composites in a relational database: as abstract data type arrays, by means of built-in array types, and as nested relations.

All the features we have studied could be added to current relational database systems, and would be useful not only for CAD but also for other applications involving spatial and highly structured data.

Contents

CHAPTER 1

Introduction

Recently there has been considerable interest in extending the use of relational database systems beyond their traditional role in business data processing and into new application areas such as text processing [79,94], computer program management [80], computer graphics [54,59,95,96], geophysical databases [58], and Computer Aided Design (CAD) [46,60]. The field of computer aided design of Very Large Scale Integrated circuits (VLSI) is in a phase of rapid development at the present time. This thesis investigates the shortcomings of relational database management systems when used to support VLSI CAD, and develops extensions to make them more useful in that role.

1.1. The Need for a Central Data Manager

In small CAD systems containing just a few programs, data management is relatively easy. Data structures can be designed for the most important programs, and long-term storage can consist of simply writing the data out to disk. The format of design files is closely connected with programs that use them, with the result that changes to programs are likely to cause changes in the format, which require changes in other programs. Examples of systems of this sort are ICARUS [23] and IGS [39].

The IGS system, as described during its development in 1978 [39] is built around an interactive graphics editor for integrated circuit layout. Data structures in memory are designed for quick retrieval of layout geometry for display on the screen. For long-term storage, a re-formatted version of the main memory data structure is copied to a file in the user's workspace. Designs are hierarchically structured and composed of small circuit "cells" and the definition of each cell is kept in a separate file. Another file, shared by everyone working on a particular design, acts as a directory for cell definitions. With this relatively simple scheme, each designer is responsible for keeping track of the parts of his or her design, and for making sure that cell names do not conflict with those used by other designers.

In larger CAD systems design data must be managed in a more systematic way, for the following reasons:

(1) There are many different kinds of data. For example, a large system might deal with circuit connectivity data, integrated circuit layout artwork, simulation results, documentation, etc., all pertaining to the same design entity.

(2) There are many different programs, which must be decoupled from each other. It is impractical to allow changes in one program or data format to force changes in other programs. A related problem is that of graceful evolution: as the system grows and new programs are added, the resulting changes should not prevent older programs from working, nor should new programs be constrained by conventions designed to suit old ones.

(3) The flow of design data must be coordinated. Many data files generated at different times depend on each other in complicated ways, and care must be taken to insure that the right information is always used for each CAD

operation. The flow may be too complicated for designers to remember and they will need help from the computer.

(4) Several designers must be able to work on the same design without interfering with each other. They must be prevented from destroying each other's work, for instance by overwriting each other's changes. On the other hand, one designer should not have to deny others access for long periods of time while he works on a design.

(5) The system must protect against loss of data due to system crashes, program bugs, etc. This is especially important for large design projects representing substantial investment in designer effort.

(6) The volume of data becomes very large. While small systems may be able to function with main memory and a few disk files, large CAD systems can be required to handle hundreds of megabytes of data [87]. This data is usually divided among many files, which must be managed in a systematic way. Very large individual files become impractical unless special techniques such as indexing are used, and casual use of virtual memory is an inefficient way to handle extremely large amounts of working data.

(7) Support for versions and revisions must be provided, so that designs can evolve gracefully over long periods of time.

(8) Naming conflicts arise when many designers produce hundreds of design entities in connection with a single project. These must be prevented or resolved as part of design management.

1.2. Data Management in Existing CAD Systems

Methods used for data management in larger CAD systems are quite varied, but for simplicity we can group such systems into three categories:

(1) Systems that rely on facilities provided by a sophisticated operating system, especially its file system.

(2) Systems containing a separate data manager built using file system access methods.

(3) Systems using a general-purpose DBMS adapted to the CAD application.

We consider each category in turn.

1.2.1. CAD Systems Based On Operating System Facilities

Probably the most popular approach is to rely mainly on a operating system and its file system. Operating system technology is mature, and systems are available that are powerful, efficient, and reliable. This approach was taken for the SCALD system [67,68], Designers Workbench [73], and the Berkeley CAD project in its original form [72].

The problem of dealing with many kinds of data and many programs is solved by making each small body of data a separate file. Typically, a program reads data from one file, does some processing, and produces another file. Use of a standard form for each kind of data decouples programs from each other. Individual programs can be changed without affecting others as long as they conform to the standards for the files they read and write.

This standardization generally occurs at two levels. First, data is usually represented as characters, which can be processed by any computer, recorded on tape in agreed-upon ways, and transmitted easily from one place to another. Second, information of particular kinds is encoded using special textual languages, for example the CIF [69] and CAESAR [75] formats for representing mask artwork and circuit design structure, and ISP [7] for describing the functioning of a computer's central processing unit. Such representations are not particularly efficient in terms of storage space or ease of encoding and decoding information. However, they are easy to use from an application programmer's point of view. An automatically generated parser [41] can help a program read an input file, and if anything goes wrong the programmer can examine the file and understand it.

Coordination of the flow of data can be accomplished by the use of protocols, along with computerized tools. Another approach relies on an authoritative supervisor program like Designers Workbench [73] to oversee all design activity. Problems with concurrent access can be solved simply by allowing only one person at a time to modify a file, or by using a locking scheme whereby a designer can check out a part of a design, thus preventing others from changing it.

As an example system from the first category we consider the Berkeley CAD project during the first phase of its development in which the aim has been to develop a "toolbox" of computerized design aids [65,72]. Each tool does a fairly small and well-defined job, and a designer uses many of them during different phases of the design process. Examples of tools are graphics editors [48,75], design rule checkers [3], programmed logic array generators [36,56], circuit extractors [27,37], electrical simulation programs [52,70], and programs to produce plots of complete designs [26].

This approach results in the production of many files containing data in several formats pertaining to numerous design entities in different stages of development. Files are managed with the help of tools that are part of the UNIX [42] programming environment. For example, the hierarchical directory structure helps to group files according to the design entities they pertain to, the SCCS (Source Code Control System) program [85] logs changes to files and archives old versions, and MAKE [24] ensures that changes in files are propagated to others that depend on them.

The Berkeley system is flexible and easy to expand and change, and has been quite successful. However, it has grown to the point where the design environment is rather confusing, because there are dozens of different tool programs and several data formats which can be converted to each other with varying degrees of success by another collection of translator programs. Generally the system works well in a university environment, where students work on fairly small projects. Even here problems sometimes arise, for example when data structures become too large to be handled efficiently in virtual memory on a VAX 11/780 computer.

The problem of mounting complexity has been addressed by the development of the SQUID data manager [50], which provides a uniform interface for access to CAD data of all kinds. SQUID works in conjunction with HAWK [50], a graphics editor that can manipulate design information in various forms, such as schematics, layout, and documentation. SQUID maintains separate files for different kinds of data relating to design entities, and makes use of virtual memory as a cache in order to provide fast access. Unification of data management under SQUID will move the Berkeley system out of the first category, CAD systems whose data management is based on operating system facilities, and into the second, which is described next.

1.2.2. CAD Data Managers Based on Access Methods

The second category contains CAD systems having special central data management facilities built on top of sophisticated file system access methods. Examples are IBM's Engineering Design System [87] and the DB-EXEC program developed for Raytheon by Ciampi and others [15], and the Berkeley system with the addition of the SQUID data manager. With this approach it is possible to tailor the database manager to the application at hand, while taking advantage of low-level facilities to manage disk space, to maintain efficient access paths, and to read and write data. However, building the data manager still requires a considerable amount of work, and if the system is too closely matched to current needs it may later constrain the ways in which the CAD system can evolve.

IBM's Engineering Design System [87] is an example of this kind of system. Its purpose was to support computer aided design activities for a new line of computers, by serving as a central database containing all logical and physical design data. All application programs would access the data in this database. This system was built using the Basic Direct Access Method ([11], cited in [87]) because the required capabilities were not available in a general-purpose DBMS when the project was begun in the late 1960's.

The database contains several groups of files built into a structure determined by the CAD application. A set of files called the CID contains a master copy of all the design data. One CID file holds data for chips, another for PC boards, and so forth for each packaging level in a design. In addition there are several work files where data from the CID is duplicated but restructured for faster access by application programs. Work files are made up from the CID by "extraction" programs, and later any changes are merged back into the CID by "feedback" programs. A protocol governs the extraction and merging of work files, to ensure that changes are only applied to the correct version of a master file, and to produce a history of the derivation of a master file. Other files contain test data generated automatically by application programs, and "rules" describing the structure of the data and relationships between data in different parts of the system.

This system is quite complex and not especially flexible. It has evolved within a particular application, and its structure reflects that use. Changes to the structure of the database appear to require major reprogramming. However, this is a production system, so changes are probably slower and more carefully considered than in experimental CAD systems like those found in universities. An example illustrates how the system has changed over time: Originally there was only one CID file for all levels of a design, but eventually it became so large that it occupied multiple disk packs. It was then broken up into several files for the different levels in the design hierarchy, that is, chip designs, board (carrier) designs, etc. This entailed restructuring the CAD database as a whole.

1.2.3. CAD Systems Using a General Purpose DBMS

In the past, CAD system builders have been reluctant to use general-purpose DBMS's for several reasons; see [20,32,46,49] and especially [88]. They may be unavailable, as was the case when IBM's Engineering Design System was begun, or they may not provide needed facilities, or they may be too slow. Often preexisting CAD programs are incompatible with the use of a central database system, because they are designed to handle data storage themselves by means of files in particular formats. In addition, the design engineers who develop CAD programs may be unfamiliar with database technology. However, using a general-purpose DBMS in a CAD system can have important advantages, namely:

Centralization. Use of a central database subsystem offers better control over the data of an enterprise like a design project. It is easier to keep track of where the data is, to control access to it, to manage versions, and to maintain integrity and consistency.

Specialization. Application programmers are relieved of most of the work of data management, and are able to concentrate on other aspects of their programs. The DBMS is developed, debugged, documented and maintained by someone else. Zintl considers this to be the most important advantage of using a general-purpose DBMS [102]. In addition, the developers of a DBMS specialize in data management and are able to apply more sophisticated techniques than application programmers can justify taking the time to learn.

A number of projects have been undertaken to build CAD systems around general-purpose database systems, for example those described by Korenjak [55], Roberts [83], and Zintl [102]; see also [6]. As an example we consider the system built by Zintl and others at Siemens. The implementors saw the use of a general-purpose DBMS as being worthwhile, but knew that a considerable effort would be required to make it work, since it did not immediately fill the requirements of their CAD application. Good experience with a previous version using a network DBMS prompted them to use Siemens's own network system, called UDS. They recognized several difficulties with using a network system, namely:

(1) Only one copy of the data would be stored, and the structure chosen would not be ideal for all uses of the data.

(2) Changing the data structures in a network database would be difficult, and would involve changes in programs using the database.

(3) Data is presented one record at a time, but application programs usually need to use much larger blocks of data.

(4) They feared that the cost of administrative functions like concurrency control and data integrity checking would slow the system down severely.

They have attacked these deficiencies by interposing a special "database management program" (DBMP) between application programs and UDS. Client programs communicate with the DBMP by sending commands in a "high level language" designed especially for the CAD system, and the DBMP translates these commands into statements of the data manipulation language of UDS. Application programs are insulated from changes in the database storage schema, because the DBMP can be modified to track the changes and application programs need not change their high level commands. In this way the DBMS can be tuned for good performance by changing storage structures and access paths without affecting application programs. The data structures still cannot be made ideal for all uses of the data, but the implementors feel that this is not a serious problem.

The DBMP also offers client programs a many-records-at-a-time interface. For instance if a high level command asks for the retrieval of all records fitting a certain description, the DBMP issues a separate command to UDS to retrieve each record and gathers them up, finally returning them to the client program all at once.

In addition, the DBMP serves to regulate concurrent access to a database. All users of a database send their commands to a single DBMP, which serializes their requests and processes them one at a time. Use of the single-user version of UDS avoids the overhead associated with concurrency control in the multi-user version, which was available. This scheme does not appear to provide for extended "conversational transactions" [60] of the kind needed for long sessions of interactive design work, however.

1.3. Using a Relational DBMS as a CAD Data Manager

It is our thesis that a relational database management system can be adapted to work very well for CAD, and in fact at least two projects are presently underway to use general-purpose relational systems for CAD databases [46,53,62]. Such use offers the following advantages, in addition to those that come with using any kind of central DBMS:

Ease of use. All CAD programs can access data in a relational database by means of a high-level non-procedural language that provides a set-at-a-time interface [2,4,81]. Client programs specify the desired data by giving relevant values and qualifying predicates, not by navigating through data structures. Therefore the programs need not know addresses of stored data or even details of the storage structures. The database system retrieves data values directly into program variables, and takes values from variables for storage. There is no need to parse a language to extract data values.

Performance. A relational DBMS can provide much more direct access to individual data items than is possible with text files. If a collection of data is represented by means of a text string in a special language, with individual items separated by newlines and preceded by keywords, then the only way to search for a particular item is to scan the string until it is found. A relational DBMS stores data items as separate entities, and can build indexes on them. Individual items or sets of items can be retrieved immediately without processing the rest of the file. A similar principle applies to larger data objects: a data manager based on a file system must search through directories to find design objects, whereas a relational database system can register them in relations, and find them quickly by means of indexes or special storage structures.

A relational DBMS can provide a variety of access paths into a body of data that are fast and efficient, and that can support special kinds of access, such as searching over a range of numeric values. Such access methods include hashing, extendible hashing [22], ISAM indices, and B-trees [5]. More than one access path can be provided for the same data, favoring different kinds of access. All these structures can be set up easily, by specifying to the DBMS what structure is desired, and are maintained automatically as the database changes.

Data Independence. A non-navigational, set-oriented access language like QUEL [34,99] or SQL [4] presents data to a DBMS user as just tables of values, and hides the access paths and storage structures of relations. In addition, a *view* mechanism [14,90] allows a program to see data in a different form from the way it is stored. It can even present the same data to different programs with different apparent structures.

There are two important consequences. First, changes in the database structure usually do not force changes in working programs, thus making it easier to keep programs working in the presence of inevitable change. Second, individual programs can view data in a way that serves them best. They are not constrained to represent their data values in a form that was once ideal for another purpose, or use a lowest-common-denominator "standard" like CIF [69].

Consistency and Integrity. Views also make it much easier to keep data consistent, since a single copy of the data accessed through multiple views takes the place of several stored files containing different forms of the same data. Even constraints that span across different parts of the database can be enforced with an "integrity constraint" facility built into some DBMS's [90]. This feature is used in business applications to enforce rules like "no employee's salary shall be greater than their manager's," but could equally well insure that "no polysilicon feature shall be narrower than 2 lambda."

Concurrency Control. Because several people often work on a design at the same time, concurrent access is important in CAD. Banking-style transactions and locking as normally provided in a general-purpose DBMS are not ideal for CAD applications [88], but locking schemes are well developed and may form a basis for new concurrency control mechanisms. Others are currently working on this problem [61,62,92].

Recoverability. One way to insure that previous states of a design can be recovered after data has been lost because of system crashes, program bugs, and user mistakes is to periodically dump selected files to tape. This amounts to taking snapshots of parts of the database at scheduled times. Restoring destroyed files usually involves a manual step, in which saved versions are read from tape. Classical banking database systems provide automatic logging of old and new data values for each transaction, which usually involves only a few updates, and automatic recovery from failed transactions. This approach is potentially more efficient because it makes saving of database state dependent on actual update activity rather than a fixed schedule.

Current relational systems are not ideal for CAD use, and have a number of weak points, for example the following.

Slow speed. Because of their general-purpose nature, relational DBMS's perform specific data manipulation tasks much more slowly than specially written programs. They are also slower than other kinds of database systems that require the accessing programs to use information about the storage structure of the data. Thus the decision to use a relational system sacrifices speed on specific tasks in order to avoid building and maintaining special programs for all required data management jobs, and to isolate CAD programs from data storage structures.

Awkward interface. Users of a relational database system must learn yet another language, the one presented to them by the DBMS. Furthermore, the features offered by relational DBMS's tend to be better suited to banking and business applications than to CAD, because database systems were developed mainly in that environment. For example, they usually provide transaction management, which allows one or a few small records to be updated safely in one quick operation. Extended editing sessions, in which a designer makes many changes to a design over a period of hours or days before updating the official design, are not well supported. This mismatch between the needs of CAD and the features provided by relational systems makes such systems frustrating to use. Furthermore it aggravates the performance problem by forcing awkward representations of design data, and by causing operations on it to be expressed in unnatural ways.

We believe that a great deal can be done in this area to make relational database systems easy to use and efficient in supporting CAD, and that is the aim of this work.

1.4. An Outline of the Investigation

The remainder of this thesis is the report of an investigation into new features for relational systems that can make them more effective for CAD use. The work was done using the INGRES database management system [34,91,99], but should also be applicable to other relational systems.

In Chapter 2 we describe a series of tests that reveal some of the weak points of a general-purpose relational DBMS when used for CAD, and suggest some specific enhancements. These are investigated further in Chapters 3 through 6. Chapter 3 contains a discussion of transitive closure operations and how to support them in a database system. Such operations are needed to expand hierarchical designs and for other CAD tasks. A new index structure called an R-tree is presented in Chapter 4. It can be used to retrieve spatial data according to its location, which is useful in CAD and other applications. DBMS users (e.g. CAD application programmers) sometimes need to set up specialized indexes for fast access to data based on special semantics, and a way to do this is described in Chapter 5. The database system calls on semantic routines provided by the user in order to obtain the information needed. In Chapter 6 we discuss several ways to allow single columns of relations to contain multiple data items, in order to represent aggregate objects consisting of collections of smaller objects. Chapter 7 is a summary of our conclusions.

CHAPTER 2

Retrieval of VLSI Design Data Using INGRES

In this chapter we describe a series of tests in which INGRES [34] performed data retrieval operations typical of computer aided design of VLSI circuits. INGRES's performance was compared against that of KIC, a graphics editor for integrated circuit designs [48]. Section 1 contains background on VLSI design, and a description of KIC. A relational database schema suitable for storing the information KIC stores in non-relational form is presented in Section 2. In Section 3 we describe three data retrieval operations performed by KIC, and INGRES test programs that do the same thing using a relational database. The results of the tests are presented in Section 4, and analyzed in Section 5. Finally in Section 6 we offer a list of proposals for improving INGRES's performance.

2.1. Background on VLSI Design

An integrated circuit is constructed by a photographic process using a set of *masks* describing the *layout* on each of several processing layers. The part of the design process we are concerned with is the definition of the layout patterns for a circuit. Because VLSI circuits are very complex, containing thousands of transistors, it is impossible to design one as a single homogeneous collection of layout features. The design problem must be broken down into smaller parts, and standard practice is to build large designs from relatively simple circuit "cells". Each cell can contain layout geometry in addition to subcell references, which specify instances of other cells used as components. The design of a particular cell is stored only once, although it can be used as a component any number of times in other cells [23,39,48,67,69,75].

A completely expanded design takes the form of a tree of cell instances, with a single cell at the root. The root has subcells, which make up the next level of the tree; the subcells have subcells, and so on. Each level of the tree consists of instances of subcells used as components in the level above. At the leaves are cells consisting of layout geometry only, and containing no subcells. A design tree can contain multiple instances of the same cell. The design tree for a typical VLSI circuit is between 5 and 15 levels deep and contains hundreds of cell instances. The tree for the RISC BLUE computer [44], for example, is 10 levels deep.

Figure 2.1 shows a small circuit cell called Cell-A, and below it Cell-B, which is hierarchically structured. Cell-B contains two instances of Cell-A, in addition to the vertical bars and connecting stubs along the sides, which are part of Cell-B itself. Layout geometry on different layers is indicated by different fill patterns.

The KIC graphics editor [48] is an important VLSI design tool at Berkeley. It displays the layout for a cell on the screen of a color graphics terminal, and allows the designer to add and remove pieces of layout geometry and place subcells interactively. Different processing layers are represented on the screen by different colors. Part of the screen works as a "magnifying glass", which shows a small section of the layout enlarged for easier editing. The designer chooses editing functions from menus along the left and bottom edges of the screen. He or she operates the editor by typing on a

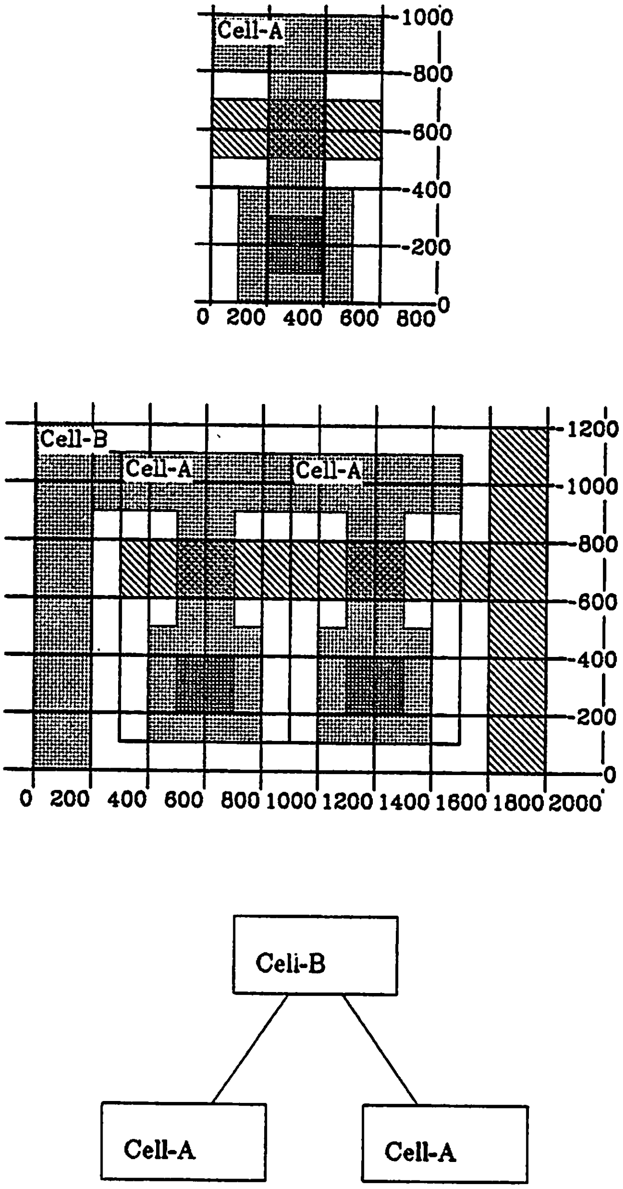


Figure 2.1: A Hierarchically Structured Circuit Cell

keyboard, moving a cursor on the screen by using a tablet and mouse, and selecting items by pushing buttons on the mouse. KIC can show subcells of the cell being edited either as outline boxes or in full detail. Many instances of the same kind of cell can appear on the screen at the same time. Changes to the definition of such a cell will be reflected in all its instances.

Between editing sessions KIC stores cell descriptions as text files in extended CIF format [69]. Each file contains specification of layout geometry and subcell calls, which are references to other cells. Figure 2.2 shows KIC's text versions of Cell-A and Cell-B. Lines beginning with "L" specify the layer for layout geometry that follows, for instance "ND" stands for the diffusion layer in the NMOS circuit technology. Lines beginning with "B" describe "boxes" or rectangles, giving the x- and y-

```
(Symbol Cell-A);
9 Cell-A;
DS 0 1 1;
L ND;
B 600 200 300 900;
B 200 400 300 600;
B 400 400 300 200;
L NP;
B 600 200 300 600;
L NC;
B 200 200 300 200;
DF;
E

(Symbol Cell-B);
9 Cell-B;
DS 0 1 1;
9 Cell-A;
C 0 T 900 100;
9 Cell-A;
C 0 T 300 100;
L ND;
B 100 200 250 1000;
B 200 1200 100 600;
L NP;
B 100 200 1550 700;
B 200 1200 1700 600;
DF;
E
```

Figure 2.2: KIC's Text Representation of Circuit Cells

coordinates of their centers and their widths and heights. Coordinates are given in KIC units, which are 1/100 of the smallest increment of measurement used for layout. In the file describing Cell-B, subcell calls are indicated by lines starting with "C". Each "C" line tells where to position a subcell, giving its x and y displacement, and the preceding "9" line gives the name of the subcell.

At the beginning of an editing session, KIC reads the text files for the selected cell and all its subcells and builds another representation of their layout in program memory. The internal representation is a pointer-linked structure using bins which allows the program to reach the right data quickly during editing operations. For example, KIC can retrieve pieces of layout geometry selectively according to their layer, the cell or subcell they belong to, and their approximate location on the screen. The program manipulates the internal form during editing, updating the text file only if asked to do so by the user. At the end of the session KIC composes a new text version from its internal data structures and stores it as a new file, or alternatively overwrites the old file. See [12] for a detailed description of KIC's data structures and operation.

2.2. A Relational Design Database

In order to compare INGRES's performance with KIC's, it was necessary to load a relational database with the same design data as KIC would use. The first step was to design a database schema capable of representing the same information as in a collection of KIC files. Our database has the following five relations, which are explained below:

```
cell-master (name, author, master-id, defined)

box (owner, use, x1, x2, y1, y2)

wire (owner, use, wire-id, width, x1, y1, x2, y2)

polygon (owner, use, polygon-id, vertnum, x, y)

cell-ref (parent, child, cell-ref-id,
        t11, t12, t21, t22, t31, t32).
```

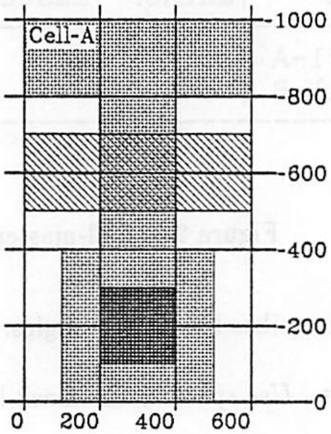
The *cell-master* relation binds the external names of cells to identifying numbers that represent them within the database. The external name of a cell is the name known to the human user, e.g. "Cell-A." The internal identifying number is unique for each cell, and is used only within the database. Using i.d. numbers this way has a number of advantages over referring to a cell by its external name; see [17,60]. For example, if two cells are accidentally given the same external name, the designs can be kept separate, and the ambiguity can be resolved later if necessary by renaming one of them. I.d. numbers are often more compact than textual names, which is advantageous for references within the database.

Name is the external name of a cell and *author* is the name of the person who designed it. *Master-id* is the unique identification number assigned to the cell. The *defined* field is used only during loading of the database, to distinguish between cells whose data is really in the database, and those that have been referred to, and assigned i.d. numbers, but whose data has not yet been loaded. Figure 2.3 shows tuples of the *cell-master* relation for the example cells Cell-A and Cell-B.

```
|name     |author |master_id |defined |
|-------------------------------------|
| Cell-A  |toni   |         1|y       |
| Cell-B  |toni   |         2|y       |
|-------------------------------------|
```

Figure 2.3: Cell-master Relation

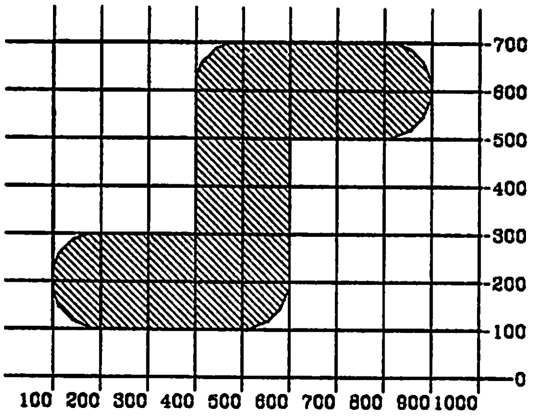
The *box* relation describes layout rectangles. *Owner* is the identifier of the cell of which each box is a part. *Use* specifies the mask layer, e.g. ND for NMOS diffusion, as in a KIC file. *X1* and *x2* are the x-coordinates of the left and right sides of the box and *y1* and *y2* are the y-coordinates of the top and bottom. Figure 2.4 shows Cell-A's layout and gives the corresponding *box* tuples. Other tuples, not shown, would represent layout rectangles for Cell-B and other cells in the database.



owner	use	x1	y1	x2	y2
1	ND	0	800	600	1000
1	ND	200	400	400	800
1	ND	100	0	500	400
1	NP	0	500	600	700
1	NC	200	100	400	300

Figure 2.4: Box Tuples for Cell-A

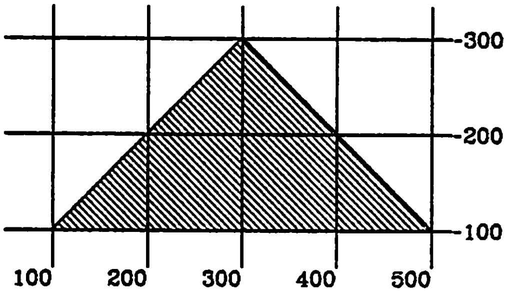
A "wire" is a set of lines used to make electrical connections between points in a circuit. Each tuple in the *wire* relation describes one line segment, giving the coordinates of its centerline *(x1, y1, x2, y2)* and its *width*. *Wire-id* is an identifying number used to group together segments belonging to the same wire. *Owner* and *use* mean the same as for the *box* relation. Figure 2.5 shows and example of a wire and the tuples of the *wire* relation needed to represent it.



```
|owner  |use  |wire_id  |width  |x1    |y1    |x2    |y2    |
|------------------------------------------------------------|
|      1|NP   |        2|    200|   500|   600|   800|   600|
|      1|NP   |        2|    200|   500|   200|   500|   600|
|      1|NP   |        2|    200|   200|   200|   500|   200|
|------------------------------------------------------------|
```

Figure 2.5: An Example Wire

A "polygon" is a solid shape with any number of vertices. One vertex is stored in each tuple of the *polygon* relation. *X* and *y* are the coordinates of the vertex, and *vertnum* orders the vertices (tuples) within one polygon. Vertices of a single polygon are grouped together by having the same value in the *polygon-id* field. Figure 2.6 shows an example polygon (a triangle) and the corresponding tuples in the *polygon* relation.

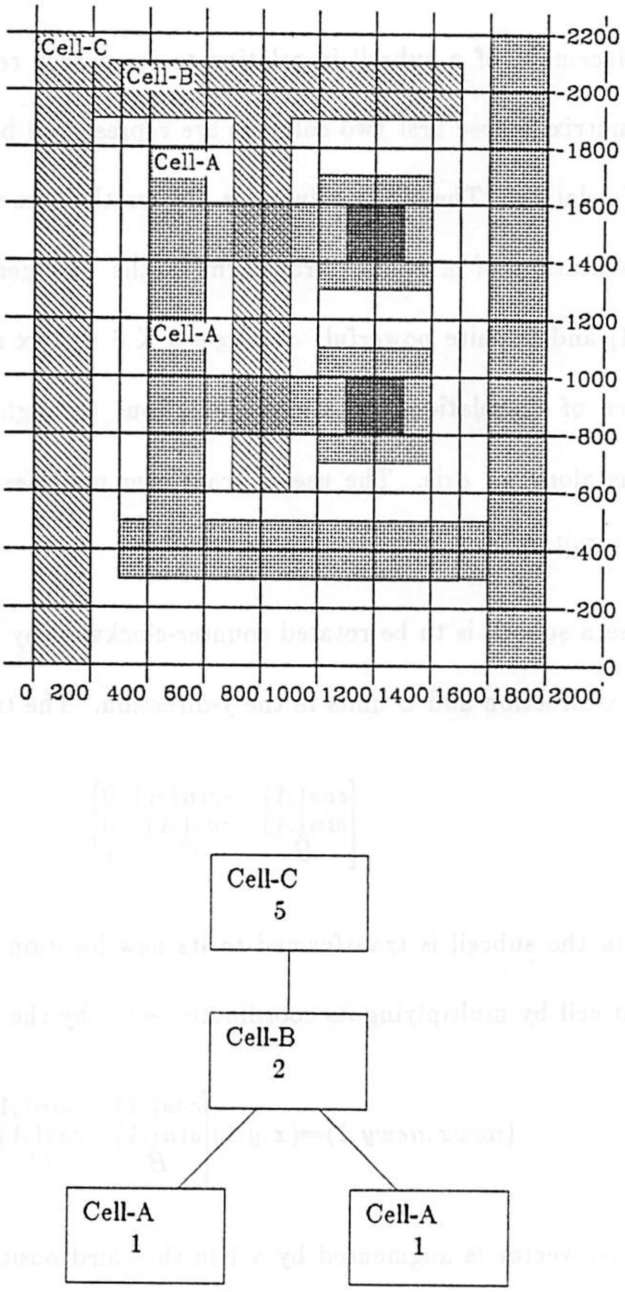


```
|owner |polygon_id |vertnum |use |x    |y    |
|---------------------------------------------|
|     3|          4|       1|NP  | 100| 100|
|     3|          4|       2|NP  | 500| 100|
|     3|          4|       3|NP  | 300| 300|
|---------------------------------------------|
```

Figure 2.6: An Example Polygon

The *cell-ref* relation describes subcell references. The *parent* field contains the i.d. number of the cell containing a subcell, and the *child* field contains the i.d. number of the subcell. Figure 2.7 shows a hierarchical cell design and the corresponding *cell-ref* tuples. The first two tuples indicate that Cell-A (i.d. number 1) is used twice as a subcell in Cell-B (i.d. number 2). Similarly, the third tuple represents the use of Cell-B in Cell-C (i.d. number 5).

The *cell-ref-id* field holds a unique identifier for each subcell reference, similar to the identifier for each cell. This field was included in the relation during an early phase of the schema design, but eventually was not needed. It is mentioned here only because it was present during the tests; it increased the size of the tuples and possibly affected the results of the tests.



```
|cell_ref_id |parent |child |t11 |t12 |t21 |t22 |t31    |t32   |
|---------------------------------------------------------------|
|           3|      2|     1|  1.|  0.|  0.|  1.|   900.| 100.|
|           4|      2|     1|  1.|  0.|  0.|  1.|   300.| 100.|
|           6|      5|     2|  0.|  1.| -1.|  0.|  1500.| 300.|
|---------------------------------------------------------------|
```

Figure 2.7: Subcell References

The placement of a subcell in relation to its parent cell is specified by a 3 X 3 transform matrix, whose first two columns are represented by fields *t11* through *t32* in the *cell-ref* relation. The third column is always the same and need not be stored. This representation of a spatial transform is the one generally used in computer graphics [71] and is quite powerful. A single 3 X 3 matrix can duplicate the effect of any sequence of translations, rotations, reflections through a line, and stretches or compressions along an axis. The use of transform matrices is outlined briefly below. See [71] for a full discussion of the topic.

Suppose a subcell is to be rotated counter-clockwise by angle A, and translated B units in the x-direction and C units in the y-direction. The transform matrix is

$$\begin{bmatrix} cos(A) & -sin(A) & 0 \\ sin(A) & cos(A) & 0 \\ B & C & 1 \end{bmatrix}$$

Each point in the subcell is transformed to its new location in the frame of reference of the parent cell by multiplying its coordinate vector by the transform matrix:

$$(newx,newy,1)=(x,y,1)\begin{bmatrix} cos(A) & -sin(A) & 0 \\ sin(A) & cos(A) & 0 \\ B & C & 1 \end{bmatrix}$$

The coordinate vector is augmented by a 1 in the third position to make it match the matrix. The point can be transformed again in order to locate it in the frame of reference of a still another cell, as is necessary when subcells are nested.

As an example, consider the hierarchical design shown in Figure 2.7. Cell-C contains an instance of Cell-B, which is rotated 90 degrees and translated to the point (1500, 300). The transform matrix for this subcell reference is

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 1500 & 300 & 1 \end{bmatrix}$$

Likewise, Cell-B contains two instances of Cell-A. The first instance (the lower one in the figure) is located at the point (300, 100) within Cell-B and is not rotated, so its transform matrix is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 300 & 100 & 1 \end{bmatrix}$$

The point located at (100, 0) in Cell-A, marked with an X, is transformed to the frame of reference of Cell-B by the calculation

$$(400, 100, 1) = (100,0,1) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 300 & 100 & 1 \end{bmatrix}$$

It can be transformed again to the frame of reference of Cell-C by multiplying on the right by the appropriate matrix:

$$(1400, 700, 1) = (400, 100, 1) \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 1500 & 300 & 1 \end{bmatrix}$$

or, combining the two matrices,

$$(1400, 700, 1) = (100, 0, 1) \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 1400 & 600 & 1 \end{bmatrix}.$$

Arrays. KIC employs a compact representation of arrays of identical subcells in one or two dimensions. An equivalent feature could have been included in the relational schema by the addition of extra fields to the *cell-ref* relation, but this was not done because it would have complicated the tests we wished to perform. Arrays of subcells are represented in the relational database by multiple *cell-ref* tuples.

2.3. Data Retrieval Tests

Three data retrieval operations, corresponding to different ways KIC retrieves layout data for display on its graphics screen, were selected as being representative of CAD data handling. A test program was written to use INGRES for each retrieval operation, and the retrieval speed was compared with KIC. The test programs were written in EQUEL [2,99], which is INGRES's query language QUEL [34] embedded in C [51]. They retrieved design data from a relational database that had been loaded with data from KIC files. Each of the three tests is explained below, and the EQUEL programs are included as Appendix A.

KIC groups rectangles by processing layer before transmitting display commands to the graphics terminal, in order to minimize the number of "change color" commands. (Recall that different layers are represented on the screen by different colors.) The EQUEL test programs retrieved each layer separately in order to mimic KIC's behavior; this entailed a considerable amount of extra processing.

During normal operation KIC displays layout geometry by sending commands to a graphics terminal over a serial communication line, and transmission time is comparable to data retrieval time. Because the tests were meant to measure retrieval performance, not transmission speed, the KIC and INGRES tests were set up so that they did not transmit data to a terminal. KIC's transmission was turned off, and the EQUEL programs simply loaded data values into program variables, as can be seen in the queries.

The present tests should be taken as only a very approximate comparison between KIC and INGRES, and are probably biased in favor of INGRES. They compare a fully-developed editor containing all necessary features with test programs that performed only basic retrieval operations. For example, in the tests KIC clipped geometry to fit in the viewing area on the screen, and this involved extra computation that was not done by INGRES.

For our test data we used circuit cells from two design projects at Berkeley. GORDA is a prototype layout for a switched capacitor filter [31], and dec0-1 is a part of the RISC I VLSI computer [78], see also [44]. Each design was divided into a number of cells; there were 205 cell definitions in all. KIC files were available for these designs and were used in the KIC tests. The INGRES database was loaded from the same files by a translator program.

Table 2.1 gives the sizes of the relations in the INGRES database and their storage structures. Each relation is stored hashed on the field most used for retrieval, in order to give fast access. For example, during a query to display a cell, INGRES would need to retrieve *box* tuples with the cell's i.d. number in the *owner* field. Therefore, the *box* relation was stored hashed on that field.

Top-Level Geometry Retrieval. The first test program retrieved layout geometry data associated with a given circuit cell, not including geometry belonging to subcells. This corresponds to KIC's "unexpanded" display in which layout geometry belonging to the cell itself is shown in full detail, but subcells appear only as outline boxes. The

Table 2.1: Experimental Database

Relation	Number of Tuples	Disk Pages	Storage Structure
cell-master	205	21	hashed on name
cell-ref	1918	195	hashed on parent
box	3037	211	hashed on owner
wire	652	55	hashed on owner
polygon	1648	98	hashed on owner

operation consisted of retrieving all the *box, wire* and *polygon* tuples with the selected cell's i.d. number in the *owner* field.

Geometry Retrieval with Tree Expansion. The second test program retrieved all geometry in a fully expanded design tree. This corresponds to a KIC display of a complete design with all component cells shown in full detail.

To perform the expansion the EQUEL test program traversed the design tree breadth-first, going downward from the root one level at a time. At each level it retrieved all layout geometry belonging to subcell instances there and transformed it to its correct location with respect to the design as a whole. Then, referring to the *cell-ref* relation, it collected the i.d. numbers of all subcells of cells at the current level. These new subcells constituted the next lower level in the tree. Once i.d. numbers were collected and transforms were calculated for the new subcells, the old set was discarded and processing continued with the next level.

The test program calculated transform matrices for subcell instances according to the method outlined above and in [71]. A new transform was calculated for each subcell instance by multiplying together the transform of the current parent cell and the transform in the *cell-ref* tuple calling for the new subcell. Transform calculations were expressed in EQUEL and were performed by INGRES as part of the retrieval operations, not by the C-language part of the test program.

Retrieval by Location. The third test program retrieved top-level geometry that overlapped a small area in the middle of a cell. A similar operation is performed by KIC in order to display a small "window" in the middle of a circuit. Numerous other CAD tasks require similar access to data by spatial location, for example layout rule checking and automatic circuit compaction. (Layout rule checking is described in [3], circuit compaction in [38].) In the INGRES test program, the windowing was expressed by an EQUEL qualification clause.

2.3.1. Test Results

In the following tables *Geometry Tuples* refers to the number of tuples representing layout boxes, wires, and polygons retrieved from the INGRES database during each operation. *CPU Seconds* and *Elapsed Seconds* were found using the C language *times* routine, which obtains timing information from the UNIX operating system [42]. *Relative Time* is the ratio of the time taken by the test programs relative to the time taken by KIC.

2.4. Discussion

In Tables 2.2 and 2.3, INGRES used about three times as much CPU time as KIC. This can be attributed primarily to the fact that INGRES is a general-purpose database system, while KIC includes only those functions that it requires. Any DBMS used in place of special purpose code will show some decrease in performance.

Table 2.2
Retrieval of Top Level Geometry

Circuit	GORDA	
	KIC	INGRES
Geometry Tuples	592	592
CPU Seconds	7.5	24.0
CPU msec/Tuple	13	41
Relative CPU Time	1	3.2
Elapsed Seconds	7.5	41
Elapsed msec/Tuple	13	69
Relative Elapsed Time	1	5.5

Table 2.3
Tree Expansion with Geometry Retrieval

Circuit	GORDA	
	KIC	INGRES
Geometry Tuples	12,779	12,779
CPU Seconds	128.3	443.5
CPU msec/Tuple	10	35
Relative CPU Time	1	3.5
Elapsed Seconds	128.3	649
Elapsed msec/Tuple	10	51
Relative Elapsed Time	1	5.1

Table 2.4
Retrieval of Geometry in a Small Area

Circuit	dec0-1	
	KIC	INGRES
Geometry Tuples	448	448
Geometries in Window	87	85
CPU Seconds	.75	14.5
CPU msec/Tuple	9	171
Relative CPU Time	1	20
Elapsed Seconds	.75	33
Elapsed msec/Tuple	9	388
Relative Elapsed Time	1	45

Additionally, INGRES processes tuples with an interpreter, which is naturally slower than KIC's compiled code.

The elapsed time measurements show a larger performance difference, about a factor of five. The reason for this is the following: KIC loaded the geometry data into virtual memory data structures before the tests began, and because the computer had ample main memory and was very lightly loaded, the data apparently remained in real memory during the tests. INGRES, on the other hand, retrieved data from disk resident relations. It buffers only 15 pages of data in its own address space. Some relation pages probably remained in system buffers during each test, but were accessible only by means of calls to the operating system.

On the third test involving retrieval of data in a small window, KIC outperformed INGRES by a factor of 20 in CPU time and 45 in elapsed time, as shown in Table 2.4. One reason is that KIC has a spatial bin structure that allows it to quickly isolate geometry that falls inside the window. INGRES has no such access method and performed an exhaustive search of all the cell's geometry.

A second reason is that our schema represented single polygons by multiple tuples, one for each vertex. It was not possible to tell from a single vertex tuple whether or not the polygon intersected the window. To find the extent of each polygon it was necessary to use INGRES's *min* and *max* functions on its vertex coordinates, before testing for overlap with the window. These aggregate functions are time consuming because they require processing of extra subqueries.

2.5. Performance Enhancements for INGRES

INGRES can be made to run faster by several means, without changing the user's view of the system:

(1) By making better use of a large main memory. INGRES was originally designed to run on a machine with limited address space, and it does not take advantage of the large virtual and physical memories available on modern machines. Virtual memory provides a convenient large address space, and exploits efficient paging hardware. Direct management of a large physical address space offers the possibility of even greater performance improvement [19,29].

(2) By compiling queries [13,45]. Currently INGRES always parses queries and plans execution strategy at run time, and processes queries with an interpreter. Compiled code would run faster, and compilation could be done ahead of time for queries that were known in advance. This would often be the case when INGRES was serving as a back-end for another program.

(3) By file system enhancements. The version of the UNIX file system used for the tests has rather small pages (1024 bytes), so INGRES must make many requests for pages to be fetched from disk, each requiring an operating system call. Larger pages would help considerably. In addition, pages from a single file reside in scattered locations on the disk, so that even a sequential scan of a file generally requires disk head movement at each page boundary. Access would be faster if pages of a file were clustered near each other on disk. Also INGRES should be able to request several pages that it will need soon, and continue processing while they are being fetched. The best solution may be for INGRES to bypass the operating system's file system entirely and manage its own files.

(4) By improving communication. Efficient interprocess communication is essential if INGRES is to work well in cooperation with application programs. Our test programs communicated with INGRES through UNIX pipes, which are not especially fast. Also, internal communication within INGRES tends to be unnecessarily slow.

Such performance enhancements are part of the process of turning a prototype DBMS into a commercial product. It is appropriate to leave such development to commercial interests. Hence we decided to concentrate on designing new features that would improve performance by allowing users to request needed functions in a more compact and straightforward way, and on finding efficient ways to implement these features. We discuss in turn several such additions.

2.5.1. Transitive Closure

Our second test program expanded a design tree of indefinite height in which tuples of the *cell-ref* relation served as links between cell instances on one level and their subcells on the level below. Finding all the cell instances in the tree is related to taking the transitive closure of the *cell-ref* relation, and does not correspond to any single INGRES query. It was necessary to have the test program iterate down the tree, using a separate query for each transition from one level to the next. A transitive closure query facility would allow such a structure to be traversed efficiently by means of a single query. In Chapter 3 we present a proposal for such a facility and describe tests of a prototype implementation.

2.5.2. Access by Spatial Location

Many CAD programs, like our third test program, need to retrieve design data according to its spatial location. INGRES could do this much faster if it could use an index based on the approximate spatial locations of data objects. Then it could retrieve all objects near the area of interest quickly, and perform a final check on just a few objects, instead of on all the data. Current INGRES secondary indexes can provide a fast access path according to the values of data fields, but these values are in effect point locations in a one-dimensional space. Database systems normally do not provide mechanisms that are effective for indexing objects of non-zero size in spaces of more than one dimension, like pieces of integrated circuit layout in a two-dimensional design area. In Chapter 4 we present a new dynamic index structure called an R-tree which addresses this need and is suitable for use in database systems.

2.5.3. Abstract Indexes

Sophisticated applications like CAD systems often associate special semantics with data objects stored in the database, and it would be desirable to be able to use such semantics to speed processing of queries. As an example, the polygons represented in our test database have areas, and it might be useful to access polygons according to this attribute. However it is not appropriate for the database system to calculate areas of polygons or perform other specialized operations on data objects.

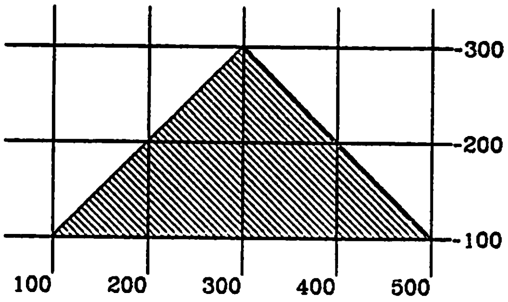
In Chapter 5 we propose an abstract index facility which allows users to define their own kinds of indexes for special purposes, and provide routines to implement the necessary semantics. The database system can invoke the user-supplied routines at appropriate times in order to plan and execute query processing strategies. This facility is an adjunct of the abstract data types already implemented in INGRES by Fogg [28] and Ong [74].

2.5.4. Nested Relations

Often it would be useful to be able to repeat a field in a relation a variable number of times, in order to represent some aggregate object. For instance, a polygon could be represented by a single tuple with repeating vertex coordinate fields, instead of multiple tuples. One way to support this is to allow a field of a tuple to take an entire relation as its value. With this addition the *polygon* relation in our database could be redefined as follows:

polygon (owner, use, vertex(vertnum, x, y)) where *vertex* is a family of special relations, one attached to each tuple of the *polygon* relation. Figure 2.8 illustrates how a polygon would be represented using the new schema. Similarly, the *wire, polygon* and *cell-ref* relations could be nested inside the *cell-master* relation.

In Chapter 6 we consider several ways to provide for array-like aggregate objects in INGRES. Language extensions needed for each proposal are discussed, as well as the range of queries that can be processed and the difficulty of implementation.



```
|owner  |use  |vertex                           |
|-----------------------------------------------|
|        1|NP  |vertnum |use |x     |y          |
|              |--------------------------------|
|              |        1|NP  | 100| 100|
|              |        2|NP  | 500| 100|
|              |        3|NP  | 300| 300|
|              |--------------------------------|
|-----------------------------------------------|
```

Figure 2.8: Polygon Stored in Nested Relations

CHAPTER 3

Transitive Closure and Design Trees

3.1. Introduction

Consider a binary relation R defined on a set S:

$$R \subseteq S \times S$$

The *transitive closure* of R, denoted by $R+$, is another binary relation derived from R as follows:

$$\begin{aligned} R+ = \{(x,y) \text{ such that there exist } & a_0, \ldots, a_k \in S \text{ for some } k \geq 1 \text{ with} \\ & x = a_0, \\ & y = a_k, \\ & (a_0,a_1) \in R, \\ & (a_1,a_2) \in R, \\ & \ldots, \\ & (a_{k-1},a_k) \in R \ \} \end{aligned}$$

The *reflexive-transitive closure* R^* differs from $R+$ in that it also includes pairs of the form (x,x) for all $x \in R$. For a more rigorous discussion of transitive closures see [1] or [33].

For example, let S be the set of all people, and define R to be the parent-child relation: for any two people a and b, $(a,b) \in R$ if and only if a is a parent of b. Then $R+$ defines the ancestor-descendant relation, i.e. for any two people x and y, $(x,y) \in R+$ if and only if x is an ancestor (or a parent) of y. R^* extends the definition of "ancestor" to mean that every person is his or her own ancestor.

In CAD and in other database applications the need often arises to find the transitive closure of a relation. For example, data describing electrical connections between pairs of nodes may be available, and we may need to find all the nodes connected, directly or indirectly, to a certain point. As another example, a hierarchical design may describe each component in terms of its subcomponents, and we may want to know all the parts used in a complex design at any level.

In this chapter we discuss additions to a relational database system that would allow it to process composite queries involving calculation of transitive closure in conjunction with other operations. In Section 2 we describe a useful form for such composite queries. Two existing database systems that provide some transitive closure operations are discussed in Section 3, and in Section 4 we propose additions to QUEL [34] to support composite transitive closure queries. Section 5 contains a discussion of several processing issues. In Section 6 we analyze processing costs for expansion of design trees by means of the mechanisms outlined in Section 4, and this analysis is applied to two real VLSI designs in Section 7. Implementation of transitive closure operations in INGRES [34] is described in Section 8, with the results of a series of tests.

3.2. Composite Queries

We begin with an example of a composite query, and proceed to a more formal description. Suppose we have an employee database with the schema

EMPLOYEE (name, manager-name, salary)
CHILD (name, parent-name)

and we wish to express the query "Print the names of all children of Jones and of employees under him in the company hierarchy, along with the parent's level below Jones". The subtree containing Jones and all employees under him seems to be a well-defined and easily computable set, but in standard QUEL it is necessary to compose a separate query to refer to employees in each level of the company hierarchy. This is because each link in the chain of association between an employee and Jones must be specified:

```
range of c is CHILD
retrieve (c.name, level = 0)
        where c.parent-name = "Jones"

range of e1 is EMPLOYEE
retrieve (c.name, level = 1)
        where c.parent-name = e1.name
        and e1.manager-name = "Jones"

range of e2 is EMPLOYEE
retrieve (c.name, level = 2)
        where c.parent-name = e2.name
        and e2.manager-name = e1.name
        and e1.manager-name = "Jones"

range of e3 is EMPLOYEE
retrieve (c.name, level = 3)
        where c.parent-name = e3.name
        and e3.manager-name = e2.name
        and e2.manager-name = e1.name
        and e1.manager-name = "Jones"
            .
            .
            .
```

This is clearly an inefficient way to solve the problem. Besides, the number of such queries required depends on the height of the company tree and Jones's position in it. Alternatively a temporary relation can be used to collect employees under Jones first, and this makes computation somewhat more efficient:

```
range of e is EMPLOYEE
retrieve into TEMP (e.name, level = 0)              (1)
        where e.name = "Jones"

range of t is TEMP

append to TEMP (e.name, level = t.level+1)          (2)
        where e.manager-name = t.name

append to TEMP (e.name, level = t.level+1)
        where e.manager-name = t.name

append to TEMP (e.name, level = t.level+1)
        where e.manager-name = t.name
.
.
.

range of c is CHILD
retrieve (c.name, t.level)                          (3)
        where c.parent-name = t.name
```

This expression of the composite operation allows us to repeat the same query instead of making up a new one for each level of the hierarchy. The *level* field of the intermediate relation, introduced in query (1), does not correspond to any field of the EMP or CHILD relations. It serves to pass a parameter down from one level to the next.

The result calculated above is not precisely a reflexive-transitive closure. The original relation whose closure is being calculated consists of manager-employee pairs, and its closure would contain a pair for every employee and each of his or her superiors at any level. We are only interested in a subset, those pairs with "Jones" in the manager slot. But storing a set of pairs with the same first element is redundant;

we only need the names appearing in the employee slot. On the other hand, we must calculate and store extra information that is not in the closure relation, namely the *level* field.

Generalizing from this example, we believe that composite queries composed of the following three parts will be useful:

(1) A *basis* query, a normal QUEL query operating on relations already in the database that serves to collect a starting set of tuples (a relation) to be used in later steps. This relation may contain extra fields that are useful in the composite operation but are not strictly part of the closure relation itself. In the above example query (1) is the basis query.

(2) A *closure* query, which updates the relation created by the basis query. Logically, it is repeated until further iteration produces no new changes. Often this is an *append,* as in the example (query (2)), though it can also be a *delete* or a *replace*. This query must have at least one tuple variable ranging over its own target relation; otherwise repetition is useless.

(3) A *result* query, which refers to the intermediate relation and other database relations to produce the final output or perform the desired updates on the database. Query (3) is the *result* query in the example.

Figure 3.1 shows how information flows from one part of a composite query to another.

Our proposed extensions to INGRES and QUEL allow processing of composite queries consisting of the three parts outlined above. This provides for a more general

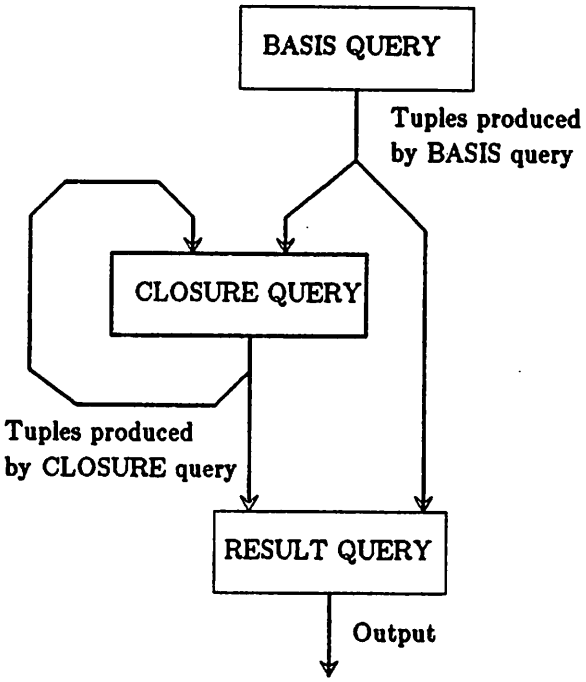

BASIS QUERY
Tuples produced by BASIS query
CLOSURE QUERY
Tuples produced by CLOSURE query
RESULT QUERY
Output

Figure 3.1
Flow of Tuples in a Composite Query

class of operations than calculating the transitive closure of stored relations, because the QUEL query at each step can perform a wide variety of operations on the data.

3.3. Existing Transitive Closure Implementations

Transitive closure operations have been implemented on at least two other database systems, Query-By-Example [104] and ORACLE [81]. Both systems can take the transitive closure of a stored relation but allow only limited kinds of other processing as part of the query.

3.3.1. Query-By-Example (QBE)

The following tables indicate how a query similar to our example would be presented to the QBE system [103,104]:

EMPLOYEE	name	manager-name
	P.Smith(5L)	Jones

CHILD	name	parent-name
	P.Sally P.Robert	Smith P.Jones

The user is shown boxes representing relations, and fills in fields of tuples in the manner illustrated. A "P" indicates that the value of the attached field is to be printed as part of the answer. Non-underlined entries are constants entered by the user, e.g. "Jones". Underlined "example items" function as variables, which can represent any value but which refer to the same value wherever they appear in the query. For instance the two uses of "Smith" specify an equi-join between the EMPLOYEE and CHILD relations. The entry in parentheses has a special meaning: its presence indicates that transitive closure is desired, and the number represents the "level number" in the closure calculation. In this case we want employees with Jones as manager at any level (the number 5 is underlined; it is only an example level).

The example tuple in the EMPLOYEE relation containing the constant "Jones" takes the place of our *basis* query, restricting the relation to just those tuples with *manager-name* = "Jones". QBE happens to provide for display of the tree level as a special case, but there is no general way to introduce new attributes for the purpose of passing parameters from one step in the calculation to the next, as we did with the *level* attribute in query (1) above. The function of our *result* query is filled by the "P" entries in the QBE query. Employee's names are printed along with those of their children, although this was not actually desired; it is side effect of placing a "P" where it will cause printing of the level number. The last line, asking for children with "Jones" in the parent-name field, is needed to include Jones's own children.

In short, QBE is able to calculate basic transitive closures and to use the result in another query. Attributes used in the closure calculation must always be a subset of those appearing in stored relations; it is not possible to introduce new ones to pass calculated values between steps in the process. The *level* is handled specially and can be used in limited ways, such as for finding the longest chains of derivation or identifying the leaves of a tree.

3.3.2. ORACLE

An ORACLE [81] transitive closure query takes the form

```
SELECT <field-list>
FROM <relation>
START WITH <qualification-1>
CONNECT BY <connection>
WHERE <qualification-2>
INCLUDING <qualification-3>
```

The following ORACLE statements approximate our example query:

```
SELECT name
FROM CHILD
WHERE parent-name =
        SELECT name
        FROM EMPLOYEE
        START WITH name = "Jones"
        CONNECT BY manager-name = PRIOR name
```

The outer query retrieves childrens' names, and the transitive closure query involving the EMPLOYEE relation is nested inside it. A START WITH clause selects a subset of a stored relation on which to apply the closure operation, in this case just Jones's tuple in the EMPLOYEE relation. This takes the place of our *basis* query, but is less general, because it does not allow addition of extra fields or other changes in the form of the tuples. The *level* field is missing, because there is no way to include it. The CONNECT BY clause specifies that the *name* and *manager-name* fields will be used to link one tuple with another to calculate the closure. Equality is the only form of linkage allowed. The PRIOR keyword indicates that the calculation proceeds downward from Jones through the company tree.

ORACLE also provides for two more kinds of clauses, which were not needed for the inner query in this example. A WHERE clause specifies a qualification that is applied to each tuple before it is used to link to other new tuples during the closure operation. That is, if a tuple does not satisfy the WHERE clause it is excluded from the closure along with all tuples it would bring in during later iterations. In our composite query formulation, the qualification of the *closure* query serves this purpose. An INCLUDING clause allows tuples to be used for linkage during the closure calculation (by satisfying the WHERE clause) but excluded from the final result (if they fail to satisfy the INCLUDING clause). A qualification can be added to our *result* query to obtain this effect.

3.4. Additions to QUEL for Composite Queries

3.4.1. Append*

The addition of a single new construct, *append**, allows composite queries with transitive closure to be expressed in QUEL. *Append** performs the iterative closure step; it is logically equivalent to an ordinary *append* repeated until no new tuples are produced. The previous example can be expressed as follows using *append**:

```
range of e is EMPLOYEE
retrieve into TEMP (e.name, level = 0)
        where e.name = "Jones"

range of t is TEMP
append* to TEMP (e.name, level = t.level + 1)
        where e.manager-name = t.name

range of c is CHILD
retrieve (c.name, t.level)
        where c.parent-name = t.name.
```

An *append** is useful only if it has at least one variable ranging over its own result relation, so that each new tuple may make possible further appends. Otherwise it is the same as an ordinary *append*.

Simple repetition of an *append* will re-generate all tuples produced by earlier iterations, possibly along with some new ones. In theory this does not matter, because we are concerned only with sets of unique tuples, and duplicates do not appear. In practice it is easy to implement *append** in such a way that earlier processing is not re-done on each iteration if there is only one variable over the target relation. This is the case in queries that expand hierarchical designs. The problem of duplicate tuples is considered more deeply in a later section.

3.4.2. Generalized * Operator

The * operator can be applied to other QUEL commands in the same sense as with *append**, namely that the command is (logically) repeated as long as it continues to change its target relation.

Retrieve into* functions like *append** but creates an empty relation first:

```
range of e is EMPLOYEE
range of j is JONES-DEPT
retrieve* into JONES-DEPT (e.all)
        where e.name = "Jones"
        or e.manager-name = j.name.
```

The first step in processing a *retrieve* into* query, after creating the relation, is to install a starting set of tuples. This can be done by using disjunctive terms in the qualification that do not refer to tuples already in the relation, e.g. 'e.name = "Jones"' in the example above. Such terms can be processed first as if by a separate *retrieve into* query, then further iterations can be performed as if by *append**. It is not possible to introduce new attributes for the purpose of passing parameters from one generation of tuples to the next, like the *level* attribute in our earlier example. The reason is because their values in the starting set of tuples cannot be specified.

This construction requires that a variable be declared to range over the result relation before it exists (like variable j above), which is illegal in current QUEL. Extending QUEL to allow this presents no difficulty, because the relation will be defined before the variable is used in a query.

*Retrieve** (without a target relation) is equivalent to an ordinary *retrieve* because the command does not affect the data it depends on. Repetition always produces the same result.

*Replace** is equivalent to *replace* run repeatedly until there are no more changes. Again, there must be at least one variable over the relation being updated or else the effect is the same as an ordinary *replace*. *Replace** is useful for propagating changes through an indefinite number of links in a network. In the following example it is used to update the EMPLOYEE relation, lowering each employee's salary, if necessary, to that of his or her manager. Each salary that is lowered could qualify others for lowering; the process must continue until the changes propagate all the way to the bottom of the company hierarchy.

```
range of e is EMPLOYEE
range of m is EMPLOYEE
replace* (e.salary = m.salary)
        where e.manager-name = m.name
        and e.salary > m.salary
```

In some cases the semantics of *replace** are not well defined, because the results depend on the order in which tuples are updated. Consider a query similar to the above, in which each manager's salary is set equal to twice that of one of his or her employees:

```
range of e is EMPLOYEE
range of m is EMPLOYEE
replace* (m.salary = e.salary * 2)
        where m.name = e.manager-name
```

Here it is not clear which of several employee's salaries to use. This ambiguity can also occur in ordinary *replace* queries and is not caused by the * operator. An aggregate over the result relation solves the problem in this case:

```
range of e is EMPLOYEE
range of m is EMPLOYEE
replace* (m.salary = e.salary * 2)
        where m.name = e.manager-name
        and e.salary = max(e.salary by e.manager-name)
```

Aggregates over the result relation can be useful in * queries if they are re-evaluated between iterations, and this is probably the best approach. However, the fact that aggregate values can change between iterations can obscure the meaning of a query. Consider the following example, which adds tuples to the HIGH-PAY relation. At each step it inserts a tuple for the manager of the highest-paid employee currently in the relation:

```
range of e is EMPLOYEE
range of m is EMPLOYEE
range of h is HIGH-PAY
append* to HIGH-PAY (m.name, m.salary)
        where m.name = e.manager-name
        and e.name = h.name
        and h.salary = max(h.salary)
```

Each iteration may add a higher-paid employee, so that older tuples would no longer qualify for inclusion if the criterion were evaluated again. With the help of the * operator this query defines a step-by-step procedure for collecting tuples; it is not the kind of static set description that QUEL was intended to express.

*Delete** is the same as *delete* in all queries without aggregates, because no tuples can qualify on a second iteration that were not deleted during the first. With aggregates, however, *delete** can be useful. For example the following query fires all managers who have no one working under them:

```
range of e is EMPLOYEE
range of m is EMPLOYEE
delete* m where m.title = "manager"
and count(e.name by e.manager-name where e.manager-name=m.name) = 0
```

We give meaning to *define* view* by stipulating that retrieval queries applied to the view should produce the same results as if the view were first instantiated by means of a *retrieve* into* query. For example, we can define a view consisting of all employees working under Jones as follows:

```
range of e is EMPLOYEE
range of j is JONES-DEPT
define* view JONES-DEPT (e.all)
        where e.name = "Jones"
        or e.manager-name = j.name
```

Now to retrieve the data for all employees working under Jones who make more than 14k, we issue the query

```
range of x is JONES-DEPT
retrieve (x.name) where x.salary > 14000
```

The * operator is not appropriate for application to *create, destroy, modify,* or *range* statements, where repetition serves no purpose.

3.4.3. Integrated Composite Queries

As an alternative to the * operator, a special language construct can be used to express all three parts of a composite query in a single statement, allowing composite queries to be processed as single entities instead of in separate steps. For example:

```
range of q is closure
retrieve (<output-list>) where <output-qual>
        closure (<closure-list>) where <closure-qual>
        basis (<retrieval query list>)
```

Here the *range* statement uses the keyword *closure* to indicate that the variable takes on the values of tuples in the intermediate relation that will be calculated during processing of the composite query. The *basis* clause corresponds to the *basis step in a three-part composite query as described in Section 3.4.31.2,* and contains one or more retrieval queries to produce the starting set of tuples for the closure operation. The *closure* clause, corresponding to the second step, is equivalent to an *append**. The *retrieve* clause functions as the *result* step, and can refer to tuples in the *closure* relation. *Closure* and *basis* clauses can also be attached to *retrieve into, append, replace,* and *delete* statements, depending on what use is to be made of the tuples gathered in the first two steps. Composite queries can be nested by including a complete composite query in the *basis* part of another.

In the unified form our original example becomes

```
range of e is EMPLOYEE
range of c is CHILD
range of q is closure
retrieve (c.name, q.level)
        where c.parent-name = q.name
        closure (e.name, level = q.level + 1)
                where e.manager-name = q.name
        basis (retrieve (e.name, level = 0)
                where e.name = "Jones").
```

Unified composite queries have several advantages over the *append** form, especially with regard to processing efficiency. It may be possible to decrease disk space usage and response time by processing two or more parts of a composite query in parallel, as outlined in the next section. Providing the system with information on how the results of the *basis* and *closure* queries will be used gives it an opportunity to optimize data structures, in order to improve overall efficiency. A query in the unified form is also easier to write and understand, because a composite operation that a programmer conceives as a single operation can be expressed as a single query instead of a sequence of steps. Furthermore the intermediate closure relation is deleted automatically after query processing is complete, whereas with *append** the programmer must remember to remove it.

3.5. General Processing Issues

3.5.1. Data Flow and Shift of Control

The boxes in Figure 3.1 (repeated below) represent the three parts of a composite query. The three processing functions can be performed by separate modules that are fairly independent of each other, and that can take a variety of different forms. If composite queries are implemented by means of *append** then each box represents a separate query presented to the database system. If the unified form described in the preceding section is implemented, then the boxes can be subroutines within one program. They can also be separate processes running on one processor, or on multiple processors.

Composite queries can always be processed by running each process to completion and gathering the result in a database relation, then beginning the next

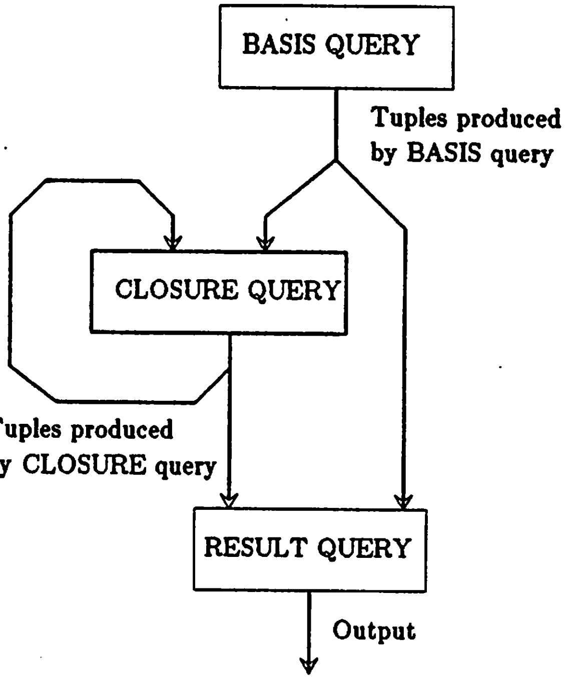


Figure 3.1
Flow of Tuples in a Composite Query

step. It is also possible to alternate between processes or run them in parallel, if the query or part of a query represented by each box refers to the output of its predecessor (backward along each arc) by means of at most one tuple variable. For example, in our tree-expansion example the feedback loop from the *closure* step to itself represented one tuple variable in an *append** referring to its own target relation. We defer investigation of the case where arcs correspond to multiple tuple variables.

If all the boxes in the diagram represent processes or sections of program running on the same processor, then it is necessary to shift control between them. This can be done according to the flow of tuples in several ways:

(1) One process can run to completion, depositing tuples in a database relation. Then the next process along one of the arcs can begin, taking its input from the stored relation.

(2) Partial results can be passed. For instance tuples can be collected in a buffer, which is passed to the next process in line when it becomes full.

(3) Tuples can be passed one by one as they are produced, for immediate use by the next process.

If the boxes represent processes running in parallel on different processors, they could run as coroutines, passing tuples to each other by means of mailboxes or queues.

Use of *append** to implement composite queries restricts all paths except the feedback loop to Option 1, because each step must run to completion before the next is begun. Using Option 1 on the feedback loop results in breadth-first tree expansion, while Option 3 gives depth-first expansion.

If the three processes are run alternately or in parallel, then it may be possible to avoid materializing intermediate relations, which may be very large. Tuples passed on and consumed by the next process along any path can be discarded immediately and not stored. In addition, response time will be improved if some data can make its way through the result box and appear as output before earlier steps have run to completion. Response time can be improved still further by overlapping CPU-intensive work in one process with I/O in another.

3.5.2. Infinite Closure

The transitive or reflexive-transitive closure of a relation containing a finite number of tuples is always finite, because it is a subset of the cross product of the sets of attribute values appearing in the original relation. However, the result of an *append** or the *closure* step of a composite query may be infinite, because new attribute values can be calculated at each step. Consider the *level-number* field in the example of Section 3.1.1. If Jones were his own manager, then the relation consisting of employees under him would contain an infinite number of tuples showing that he works for himself, with ever-increasing level numbers. Although this would not be the desired result in the example, some queries involving infinite closure may be useful. For example, such a query might be used to produce an unending stream of strings generated by a grammar. It may be difficult to predict whether the result of a given composite query is finite.

An *append** query with an infinite result will never terminate and is useless, because a subsequent query to retrieve the result can never be begun. The same is true of a three-step composite query expressed in the unified form described in Section 1.4.3 if the processing strategy involves running the closure step to completion before beginning the result query. However, options 2 and 3 above will produce an endless stream of output.

3.5.3. Duplicate tuples

Duplicate tuples may arise during processing of any of the three parts of a composite query: the basis query, computation of the closure and the result query.

Duplicates produced by the basis and result queries can be dealt with by standard methods which require instantiating the output. One such technique is to sort the relation after all tuples have been collected and remove adjacent identical tuples. Another is to store the relation in a hashed or isam structure, so that each tuple has a definite place depending on its attribute values. It is then easy to check whether each new tuple is a duplicate before it is added to the relation. The choice of a storage structure for duplicate detection depends on the number of tuples that must be stored, and this is a function of the particular query and of the data.

Duplicates produced during the closure step can be separated into several categories, and different treatments are appropriate:

(1) Duplicates may appear along a chain of tuples derived from a single starting tuple. This represents a cycle in the expansion of the closure.

(2) Several identical tuples may be siblings, i.e. they may be derived in one iteration from the same parent tuple in the closure relation.

(3) Duplicates may appear at the same distance from a tuple in the starting set (i.e. generated during the same iteration, if the expansion is breadth-first), but without having the same parent tuple.

(4) Duplicates may occur in the closure without being related in any of the above ways.

We discuss each of the above cases in turn. It is essential that Case 1 duplicates be detected, otherwise processing will not terminate. Undetected Case 1 duplicates will cause a breadth-first program to iterate indefinitely, producing again the same tuples it produced in earlier generations. Similarly, a recursive depth-first program will go into infinite recursion.

If expansion is depth-first, Case 1 duplicates can be eliminated by keeping all tuples on the derivation path from a starting tuple to the current new tuple in a duplicate-eliminating structure, such as a linked list. Each new tuple is checked agains all others in the list before it is added to the result. When the program recurses, i.e. moves downward to a new node in the tree, a new tuple is added to the list, and when it returns from a recursive call a tuple is removed. Only one such path need be instantiated at a time, therefore the space required for this kind of duplicate elimination is determined by the maximum length of a derivation chain. If the closure set corresponds to a tree, the maximum length is just the height of the tree plus one. In cases where the tree is expected to be extremely tall, then some form of search tree can be used in place of the linked list to speed the search for duplicates.

Other kinds of duplicates (Cases 2 through 4) consume extra storage space and computation time but do not affect the correctness of query processing. Characteristics of the data and the query determine whether checking for them is worthwhile. If there are no duplicates, then checking for them represents an unnecessary cost. If there are many, eliminating them can save much processing time and storage space.

Duplicates with the same parent tuple (Case 2) can be detected by collecting new tuples derived from a single parent tuple. The entire set can be discarded when all tuples derived from that parent have been generated and processing moves to a new parent tuple. Any duplicate eliminating storage structure can be used; the choice will be determined by the size of the sets expected. Duplicates within the same generation (Case 3) can be eliminated during breadth-first expansion by collecting all tuples of the current generation. In this case the space required is determined by the size of the largest generation.

Duplicates of all kinds in the closure relation can be detected by instantiating the entire closure. This approach has the disadvantage that it may require a great deal of space, and processing to find duplicates will be less efficient in a large structure. In addition, full instantiation prevents infinite closure queries from being processed at all.

3.6. Analysis of Design Tree Expansion

In this section we investigate the implications of using various algorithms and buffering schemes that are applicable to design tree expansion by means of *append**. We derive formulas for buffer space and processing time cost estimates, which will be applied to real VLSI designs in the next section. Cost estimates are based on INGRES, but should apply to other database systems with slight adjustments. The results presented in this section are relevant to other applications besides VLSI design, especially other kinds of CAD, in which trees are expanded.

3.6.1. Database Schema and Expansion Queries

Our analysis is based on expansion of hierarchical designs stored in the simplified schema, similar to that of Chapter 2, shown in Figure 3.2. Each circuit cell is assigned a unique number, *master-id* in the schema. Edges in the design tree correspond to tuples in the INSTANCE relation, which has fields for the identifying numbers of the instantiated cell and its parent, along with a 3X2 transform matrix [71] which specifies the position of the child cell in relation to its parent. The INSTANCE relation is stored hashed on the parent cell number, and we assume there are no collisions. A further simplifying assumption is that all tuples with the same parent fit on a single page, so that overflow pages are never needed. Thus if the identifying number of a cell is known, the numbers of all its children can be found with one disk access.

Pseudo-QUEL commands to expand the design whose identifying number is 1 are:

```
retrieve into TREE (cell=1, location=(0,0))
range of i is INSTANCE
range of t is TREE
append* to TREE (cell = i.child,
        location = t.location ∘ i.transform-matrix)
where i.parent = t.cell
```

The result of the expansion is produced as the TREE relation, with one tuple representing each node and containing the identifying number of a cell. We assume

CELL-MASTER (name, master-id)

INSTANCE (parent, child, transform-matrix).

BOX (owner, use, x1, x2, y1, y2)

WIRE (owner, use, wire-id, width, x1, y1, x2, y2)

POLYGON (owner, use, polygon-id, vertnum, x, y)

Figure 3.2
Schema for Design Tree Expansion

that no cell is ever used as a part within itself. This would represent an incorrect design, and could prevent the expansion query from terminating.

Only the INSTANCE relation is needed to expand the tree, although other kinds of information about circuit cells can be tagged with their numbers and stored elsewhere in the database, e.g. in the BOX, WIRE, and POLYGON relations.

3.6.2. Design Tree Parameters

In this subsection we describe several of abbreviations for numbers that characterize particular design trees. They will be useful later for compact representation of cost formulas. The reader is advised to skim this part at first, and refer back to it for explanation of parameters appearing in the formulas. Table 3.1 lists the parameters with brief definitions, and Figure 3.3 illustrates their meanings by showing their values for an example design tree.

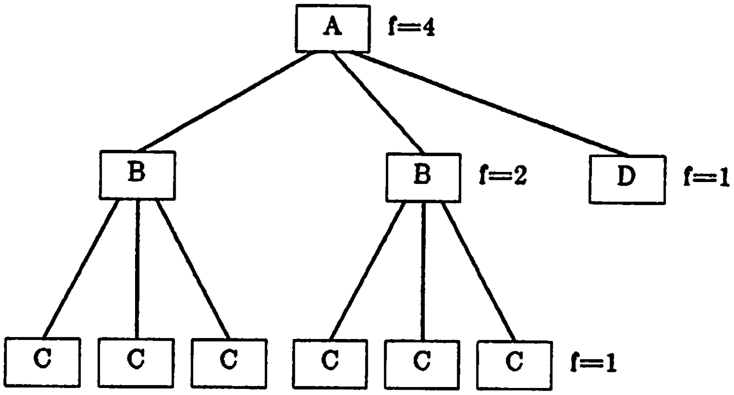
The height h of the tree is 2 as is clear from the picture. The maximum width m is 6 because of the six "C" nodes in the second level; this will be used to calculate the buffer space needed for breadth-first expansion. The total number of nodes n is the number of tuples the expansion must generate, and will be used in estimates of CPU cost. In the example n is 10.

The parameters n', n'', and i will be used to help calculate the number of INSTANCE pages that must be read from disk during tree expansion under various conditions. The value of n' is the number of nodes in the tree if we count arrays of nodes as one, but still count nodes in lower levels of replicated subtrees separately.

Table 3.1
Design Tree Parameter Definitions

h	Height of the design tree.
m	Width of the tree, i.e. the maximum number of nodes in any level.
n	The number of nodes in the tree, equal to the number of edges plus one.
n'	The number of nodes in the tree if each array of leaves is counted as one node, and in each array of subtrees the root node is counted only once.
n''	The number of nodes in the modified tree where all arrays have been collapsed to single instances.
i	The number of distinct circuit cells, equal to the number of INSTANCE relation pages used in the expansion.
k	The maximum width of the largest array-replicated subtree, with all arrays collapsed to single instances.
t	The number of TREE relation tuples that fit on one page (1024 bytes).
s	The number of subroutine-call stack frames that fit on one page.

```
$f(\text{node } x)$=  if there is an array edge above $x$ or $x$ is the root then
                1+sum($f(y)$ where $y$ is a child of $x$),
                        arrays counted as single instances
        else if node $x$ has children then
                1+max($f(y)$ where $y$ is a child of $x$)
        else
                1
```



$h = 2$
$m = 6$
$n = 10$
$n' = 5$
$n'' = 4$
$i = 4$
$k = 1$

Figure 3.3
Design Tree Parameters

This is useful for calculating the number of INSTANCE pages that must be read if buffering is used for arrays of cells but not for arrays of subtrees. In the example, n' is 5, one each for A and D, one for the array of two B's, and two for C because the two groups of three are counted separately. If INSTANCE pages used in replicated subtrees are also buffered, the corresponding number is n''. This is the number of nodes in the tree if all arrays are collapsed to single instances. In the example n'' is 4, one each for A and D, one for the array of two B's, and one for the two arrays of C's because they are in replicated subtrees.

The parameter i is the total number of different kinds of cells used in a design. Since tuples for all downward links from each cell are stored on one page, i is also the maximum number of distinct INSTANCE pages that will be needed during an expansion. In the example i is 4, for A, B, C, and D.

The parameter k refers to the maximum width of the largest replicated subtree, with arrays counted as one. We use this value to calculate the buffer space needed to save INSTANCE pages for lower levels of replicated subtrees during breadth-first expansion. Its value is 1 in the example because there is only one replicated subtree, whose width is 1 when the array of C nodes is collapsed.

Parameters t and s refer to the numbers of TREE relation tuples and subroutine call stack frames that will fit in one page of memory. Values will depend on the implementation and the tree expansion query under consideration. For the purpose of estimating space requirements we take TREE tuples to be about 30 bytes long (four for the *cell* field and four each for entries in a 3×2 *location* matrix) which gives t=30 for a page size of 1024 bytes. Stack frames are estimated to occupy about 300 bytes, and therefore s is 3.

The purpose of function f is to calculate the number of buffer pages needed during depth-first tree expansion to store INSTANCE tuples for nodes that appear repeatedly because they are in arrays or in replicated subtrees. The easiest way to describe this quantity is to consider how it would be calculated using a recursive function defined for any node in a tree. If a node is in a repeated subtree (case 1 in the definition) then f is the sum of the values of f for all its children, plus one for the node itself. This is because all nodes in the subtree will appear again. If the node is not in a repeated subtree (case 2), then the value is one plus the maximum of the child values, because the buffer space used for expansion of one child subtree can be reclaimed and used again for other children. If the node is a tree leaf (case 3) then the value is just one. The value of f at the root gives the number of buffer pages required for expansion of the entire tree.

3.6.3. Buffer Management Policies

A database system usually has a main-memory buffer with a fixed number of slots which it allocates according to an LRU (Least Recently Used) replacement algorithm. It has been suggested in [43] that a DBMS often has considerable information about page use and might be able to manage buffer space better than is possible with a blind method like LRU. (See also [86] for a discussion of buffer management in a database context.) The cost calculations below are based on strategies for managing variable-sized buffers by making use of information about the query being processed to decide when to release buffer pages.

Three buffer management policies are considered, which trade main-memory buffer space for I/O time in different ways. The first uses a minimum amount of space, and is designed to keep in memory just those pages that are currently in use, i.e. the TREE page containing the tuple for the node whose children are currently being added, the INSTANCE page with tuples describing those children, and a TREE page to receive new tuples. Buffer management consists of holding relation pages in the buffer until all tuples on them needed by the expansion algorithm have been processed, then releasing them without regard for whether they will be needed again.

The second policy involves predicting which pages will be needed again later in an expansion, and keeping them in memory. The prediction is based on the observation in [76] that circuit cells serve mainly to break a design into manageable pieces and are rarely re-used except in arrays. Arrays of identical subcells often appear in VLSI designs, and result in multiple occurrences of identical subtrees in the expanded design tree. INSTANCE pages used during the expansion of such subtrees are kept in the buffer until the array is complete. Pages used in non-array expansion are released, on the supposition that they will not be needed again. TREE pages containing new tuples just added to the relation are held until all their tuples have been examined for the purpose of adding their children.

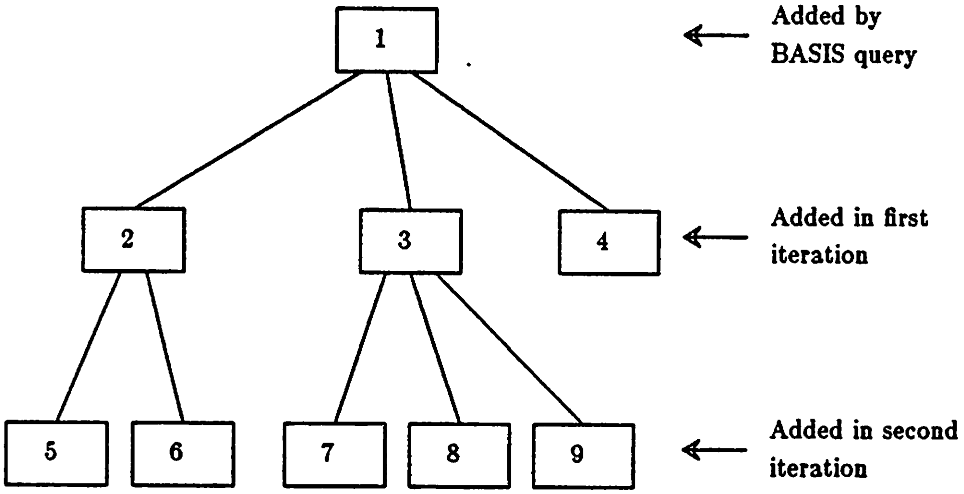
Finally we look at the case where buffer space is unlimited. TREE relation pages are handled as before, but INSTANCE relation pages used in an expansion simply remain in the buffer until the operation is finished. This insures that no page ever needs to be re-fetched.

Tables 3.2 and 3.3 have columns for each of the three policies. The formulas in the tables are discussed in the next two sections.

3.6.4. Cost Formulas for Breadth-First Expansion

In *breadth-first* expansion a design tree is produced one level at a time, working down from the root. The processing program examines tuples representing nodes in one level, refers to the INSTANCE relation to find edges going down to the next level, and composes tuples for nodes there. After one level has been completely processed work begins on the next. Figure 3.4 shows the order in which tree nodes are composed during breadth-first expansion, followed by a pseudo-QUEL program that expands a tree in this manner. Table 3.2 gives cost formulas for breadth-first tree expansion.

During expansion the TREE relation is accessed alternately in two places, as indicated in Figure 3.4.: first a tuple representing a tree node is read from the relation, then a set of new tuples is appended for the node's children. If tuples of the TREE relation are stored one after another in the order in which they are appended, the two functions can be performed by sequential scans, one to read old parent tuples, the other to append new child tuples to the end of the relation. The minimum buffer policy requires a single page for each scan. At any point during processing one page of



```
retrieve into TREE (cell=1)
range of i is INSTANCE
range of t is TREE
repeat
{
        append to TREE (cell = i.child) where i.parent = t.cell
} until no more tuples added
```

Figure 3.4
Breadth-First Tree Expansion

Table 3.2
Cost Formulas for Breadth-First Tree Expansion

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	2	$\sim m/t$	$\sim m/t$
	INSTANCE Relation	1	k	i
	Stack	0	0	0
Disk Reads	TREE Relation	$\lceil n/t \rceil$	1	1
	INSTANCE Relation	n'	n''	i
Disk Writes	TREE Relation	$\lceil n/t \rceil$	$\lceil n/t \rceil$	$\lceil n/t \rceil$
Tuples Read	TREE Relation	n	n	n
	INSTANCE Relation	$n'-1$	$n'-1$	$n'-1$
Tuples Composed	TREE Relation	$n-1$	$n-1$	$n-1$

the INSTANCE relation is needed, the one containing links between the node just read and its children.

The reading scan reads all n tuples in the TREE relation, and this requires that $\left\lceil \frac{n}{t} \right\rceil$ pages be read, where t is the number of tuples that will fit on a page. The appending scan composes all tuples but the first, representing the root, which was already in the relation at the start. All $\left\lceil \frac{n}{t} \right\rceil$ pages must be written out. Early in the expansion both scans may use the same page, which can save a few disk reads, but this effect is neglected because it will be small if the tree becomes wide quickly, as is typical for VLSI designs [76]. Buffer pages used by the read scan can be released as soon as all their tuples have been processed. Pages used by the appending scan should be written out and released as soon as they become full, unless they are also being used by the read scan.

One page of the INSTANCE relation must be read for each node including leaves in order to look for downward links, and therefore n pages reads would be required if there were no arrays. However an array construct causes the production of several consecutive TREE relation tuples corresponding to instances of the same cell. During expansion of the next level, the same INSTANCE relation page can be used to generate children of all nodes in the array. Hence the corrected number of reads is n', the number of nodes in the tree counting arrays as one. Buffer slots used for INSTANCE relation pages should be recycled immediately and used for the next INSTANCE page read.

The second buffering scheme uses more buffer space to keep pages in memory if they will be needed later. Figure 3.5 illustrates how TREE relation pages can be buffered to eliminate re-reading. The processing program works its way across one level (from left to right, say) and produces new tuples across the next lower level. When it reaches the right side of the tree, it starts again at the left side one level down. New TREE relation pages created by the appending scan are locked in the buffer and held until the read scan has finished with them, then released. Each new page may be written out at any time after the appending scan has finished filling it, possibly before the read scan reaches it. In the worst case the buffer must hold almost all of the two widest consecutive levels of the tree, but usually space for the widest single level is sufficient, and this space requirement is shown in the table as $\sim m/t$. The actual number of pages required for a given design tree varies according to the order in which children of individual nodes are generated. With this buffering scheme, TREE relation pages are never read from disk except for the first one, hence the number of disk reads is 1. They must still be written out, however, requiring $\left\lceil \frac{n}{t} \right\rceil$ disk writes.

Retaining INSTANCE relation pages can also decrease I/O. Pages used for one level of a replicated subtree are referred to repeatedly during processing of that level; they can be fetched once and kept in the buffer (see Figure 3.6). The number of buffer slots needed is k, the largest number of nodes in any level of any array-replicated subtree. The number of page reads for the INSTANCE relation is reduced to n'', the number of nodes the design tree would have if all arrays were converted to single instances.

The second buffer management strategy can be implemented by means of a *retain* flag in each TREE relation tuple to indicate whether the corresponding INSTANCE page should be held or released. The value of the flag for new TREE

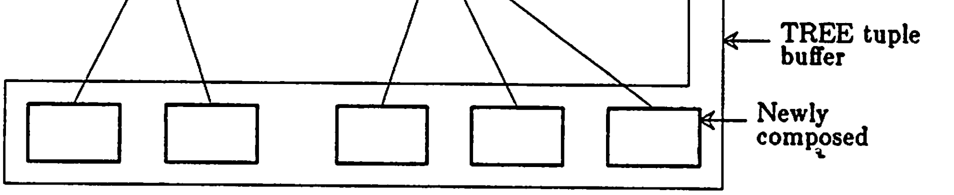


Figure 3.5
Buffering of TREE Relation Pages
for Breadth-First Expansion

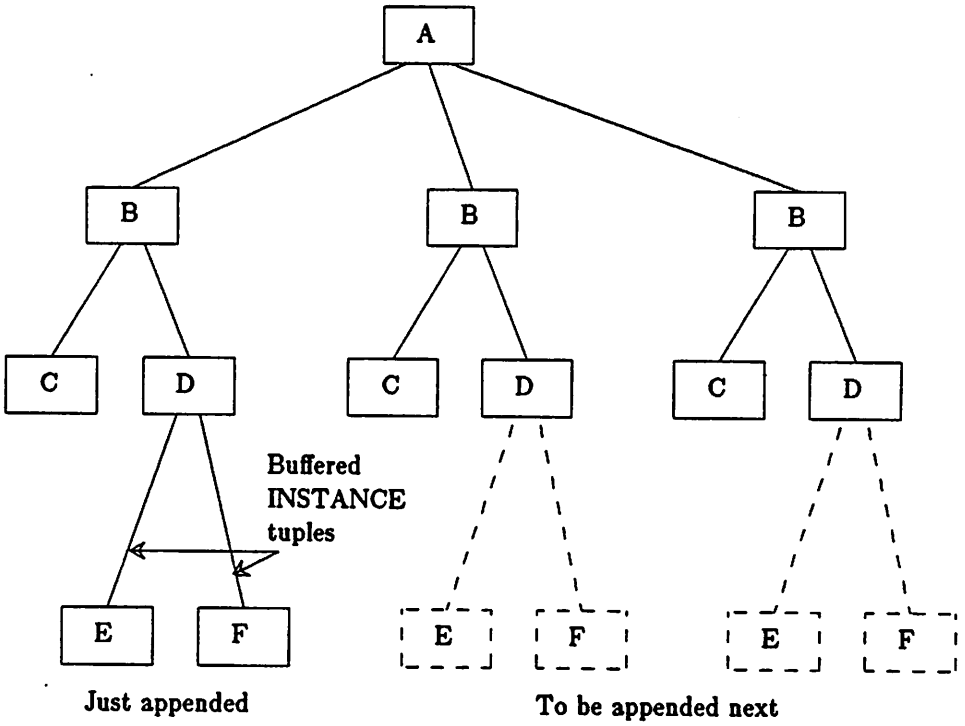


Figure 3.6
Buffering of INSTANCE Relation Pages
for Breadth-First Expansion

tuples is calculated in the following way: if the parent node has the flag set, expansion of a replicated subtree is in progress and the new child node should have the flag set also. If the parent node has the flag clear, then either it is not part of a replicated subtree, or is in the last copy. New nodes generated without an array construct should have the flag clear, and arrays of new tuples should have it set except for the last element.

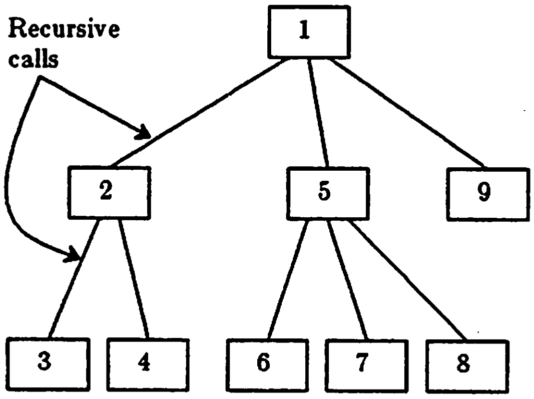
The third buffering scheme assumes that unlimited space is available. TREE relation pages can be handled the same way as in the second scheme, that is, they are held in the buffer after they are filled until they have been read. INSTANCE relation pages simply remain in the buffer until the tree expansion is complete, in case they are needed again. The number of INSTANCE pages read is equal to i, the total number of different cells used in the expansion. This is also the number of buffer slots required for INSTANCE pages.

The numbers of tuples processed is the same for all buffer management policies. The reading scan must read all n TREE tuples, and the appending scan must compose all but the root tuple, which existed at the beginning of the expansion. One INSTANCE tuple must be read in order to generate each single TREE tuple (except the root) and each array, which can be described by a single INSTANCE tuple; the total is $n'-1$.

3.6.5. Cost Formulas for Depth-First Expansion

Although the description of *append** as an iterated *append* suggests breadth-first, level-by-level processing, it is also possible to implement *append** by means of a recursive routine that expands a tree depth-first. The routine begins with the root node, refers to the INSTANCE relation to find its children, and calls itself with each child. The entire subtree dependent from each node is traversed before succeeding nodes on the same level are processed. Figure 3.7 indicates the order in which nodes are produced, and presents a pseudo-code program for depth-first tree expansion. Table 3.3 gives the corresponding cost formulas.

Arguments for the AddChildren subroutine are passed to it by placing them on the stack, which thus acts as temporary storage for TREE tuples. If the tree has height h, then $h+1$ stack frames are needed, and the space needed is approximately $\left\lceil \frac{h+1}{s} \right\rceil$ pages, where s is the number of frames that will fit on a page. The expansion



```
RootNode = (cell=1)
append to TREE (RootNode)
call AddChildren (RootNode)

subroutine AddChildren(Node)
{
        range of i is INSTANCE
        foreach (ChildNode = (cell=i.child) where i.parent=Node.cell)
        {
                append to TREE (ChildNode)
                call AddChildren (ChildNode)
        }
        return
}
```

Figure 3.7
Depth-First Tree Expansion

Table 3.3
Cost Formulas for Depth-First Tree Expansion

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	1	1	1
	INSTANCE Relation	$h+1$	$f(root)$	i
	Stack	$\lceil (h+1)/s \rceil$	$\lceil (h+1)/s \rceil$	$\lceil (h+1)/s \rceil$
Disk Reads	TREE Relation	1	1	1
	INSTANCE Relation	n'	n''	i
Disk Writes	TREE Relation	$\lceil n/t \rceil$	$\lceil n/t \rceil$	$\lceil n/t \rceil$
Tuples Read	TREE Relation	0	0	0
	INSTANCE Relation	$n'-1$	$n'-1$	$n'-1$
Tuples Composed	TREE Relation	$n-1$	$n-1$	$n-1$

routine never needs to read from the TREE relation, so only one buffer page is required, to collect new tuples as they are composed. This page can be written out and re-used as soon as it fills.

The minimum buffer management policy attempts to keep in memory all INSTANCE relation pages corresponding to TREE tuples stored on the stack, so that when the recursive routine returns from expanding one subtree, it will not have to re-fetch the INSTANCE page containing the link to the next. Thus a maximum of $h+1$ buffer slots is needed for INSTANCE pages. (See Figure 3.8.) Each invocation of the routine causes the reading of one INSTANCE relation page containing links downward from the argument node. This page is held in the buffer until the routine returns. Arrays of child nodes are handled slightly differently: the call for the first array element causes an INSTANCE page to be read as usual, but the page is not released on return until the last element of the array has been visited. Calls for the second and later elements of an array will find the correct INSTANCE page already in memory, and will uot have to fetch it from disk. Thus the total number of INSTANCE page reads is n', one for each node in the tree but counting arrays as one.

The second buffering policy takes advantage of the fact that the same set of INSTANCE pages is used to expand arrays of subtrees (Figure 3.9). This set can be kept in the buffer by passing a *retain* flag as an argument to the recursive routine. If replicated subtree expansion is in progress because of an array construct higher in the tree, this is indicated by the flag being set. The routine makes further recursive calls with the flag set and does not release INSTANCE pages from the buffer before returning. If the flag is clear, then the routine can release its INSTANCE page and all those of deeper recursive calls before it returns. In this case it sets the flag before making recursive calls for arrays, and makes non-array calls with it clear. The flag is

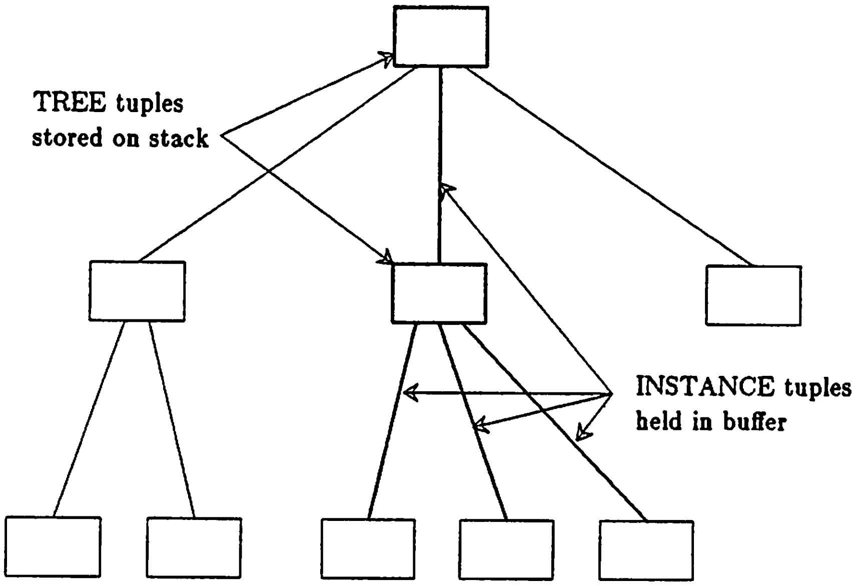


Figure 3.8
Buffering of INSTANCE Relation Pages
for Depth-First Expansion

clear for the initial call with the root node.

The function f calculates the number of buffer slots required to hold all INSTANCE pages needed to expand any replicated subtree, plus pages for the remaining links between the subtree root and the root of the entire tree. With f(root) buffers for the INSTANCE relation, the number of page reads is reduced to n'', which is the number of nodes the tree would have if all arrays were converted to single instances.

As in the case of breadth-first expansion, the third buffer management policy requires one INSTANCE page to be read for each different kind of cell used in the

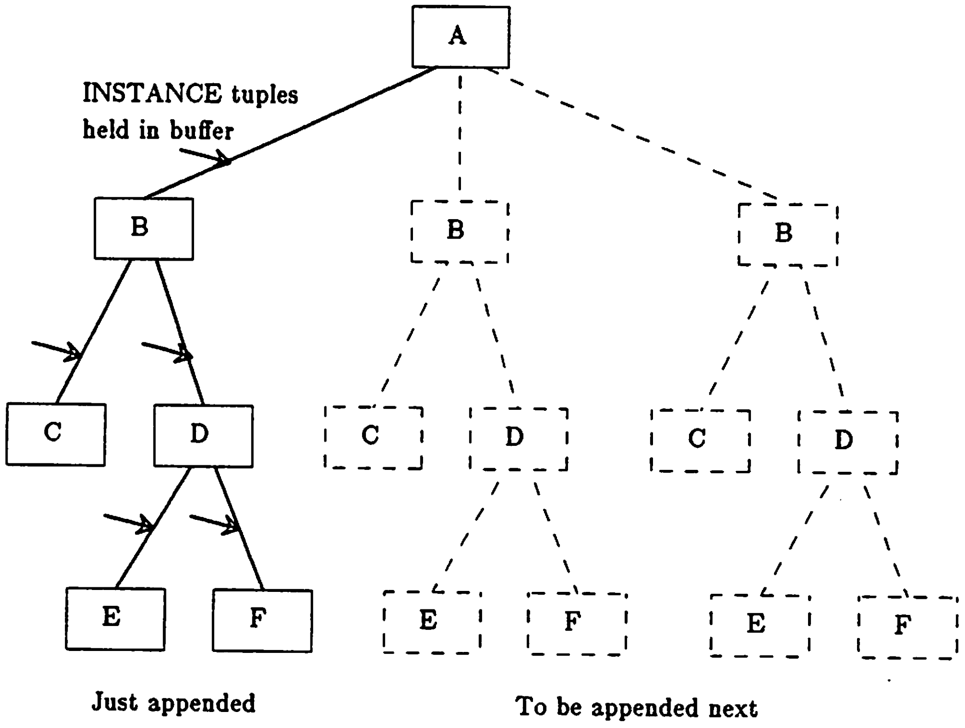


Figure 3.9
Buffering of INSTANCE Pages for Arrays
During Depth-First Expansion

expansion, and that page remains in the buffer until the process is complete. Thus a total of i INSTANCE relation disk reads are performed, and i buffer pages are required.

All buffer management policies require one page to receive new TREE tuples, which can be written out as soon as it becomes full because the recursive routine never needs to read TREE tuples. The number of INSTANCE tuples read is always $n'-1$, that is, one tuple is read for each single TREE tuple composed (all except the root), and one for each array.

3.7. Application of Analysis to Two VLSI Designs

We have applied the preceding analysis to two VLSI designs developed at Berkeley, the RISC II Instruction Cache chip and the RISC II CPU chip [44]. The cache is essentially a memory and is very regular, with a few large arrays, and contains about 45,000 transistors. The CPU design is more complex; the layout shows many irregular parts but also has large regular areas. In its finished form the CPU also contains about 45,000 transistors; it was not finished at the time of our analysis, but the few details missing should not appreciably affect the characteristics we desired to measure.

Table 3.4 shows the values of the previously defined parameters for the two designs, along with other summary information. Both trees are very wide in proportion to their height, and therefore we would expect depth-first expansion to be more efficient than breadth-first because the recursion required for depth-first expansion is shallow. The necessary buffering is for a short vertical piece of the tree, instead of an wide horizontal section as needed for breadth-first expansion.

About a third of the circuit cells are re-used in each design, contrary to our earlier assumption that cells are rarely re-used. The average number of independent uses of a cell is 3.6 for the CPU and 2.8 for the cache. This casts doubt on the wisdom of the second buffer management policy described above, though it will be seen that it still works fairly well.

Arrays are heavily used in both designs. This was observed in connection with other designs in [76] and is not surprising, because regularity is heavily emphasized in current VLSI design methodology [69]. The cache design tree contains fewer arrays than the CPU (57) but they are large, with an average size of 63. The CPU chip, a more complex and less regular design, has 235 arrays in its tree but their average size is only 12.

Tables 3.5-3.8 show the results of substituting the parameter values from Table 3.4 into the cost formulas of Tables 3.2 and 3.3. Time estimates are based on a disk access time of 30 msec. and a tuple-processing time of 3 msec., roughly what was reported for INGRES in [93].

Table 3.4
Parameter Values for RISC II CPU and Instruction Cache

		CPU	Cache
h	Height of the design tree.	10	4
m	Width of the tree, i.e. the maximum number of nodes in any level.	10063	6253
n	The number of nodes in the tree, equal to the number of edges plus one.	14050	9854
n'	The number of nodes in the tree if each array of leaves is counted as one node, and in each array of subtrees the root node is counted only once.	11465	6326
n''	The number of nodes in the modified tree where all arrays have been collapsed to single instances.	942	320
i	The number of distinct circuit cells, equal to the number of INSTANCE relation pages used in the expansion.	259	113
	The number of cells never re-used except in arrays.	167	76
	The average number of times each cell is used independently in the design.	3.6	2.8
	The number of arrays in the complete tree.	235	57
	The number of non-leaf arrays.	48	19
	Maximum array size.	1088	1408
	Average array size.	12	63
k	The maximum width of the largest array-replicated subtree, with all arrays collapsed to single instances.	9	4
$f(root)$	See Section 3.6.2.	21	7
t	The number of TREE relation tuples that fit on one page (1024 bytes).	30 (est.)	30 (est.)
s	The number of subroutine-call stack frames that fit on one page.	3 (est.)	3 (est.)

Table 3.5
Cost Values for Breadth-First Expansion of RISC II CPU

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	2	335	335
	INSTANCE Relation	1	9	259
	Stack	0	0	0
	Total	3	344	594
Disk Reads	TREE Relation	469	1	1
	INSTANCE Relation	11465	942	259
Disk Writes	TREE Relation	469	469	469
Disk Accesses		12403	1412	729
Disk Time @ 30 msec.		372 sec.	42 sec.	22 sec.
Tuples Read	TREE Relation	14050	14050	14050
	INSTANCE Relation	11464	11464	11464
Tuples Composed	TREE Relation	14049	14049	14049
Tuples Processed		39563	39563	39563
CPU Time @ 3 msec/tuple		119 sec.	119 sec.	119 sec.
Total Time		491 sec.	161 sec.	141 sec.

Several observations can be made about these results:

(1) According to the time estimates shown in all four tables, a DBMS performing design tree expansion would be very CPU bound except with a minimum size buffer.

(2) The second-level buffering scheme decreases the amount of I/O drastically compared to minimum buffering, by a factor of ten for both breadth-first and depth-first tree expansion. This reflects the fact that a large proportion of the nodes in a design tree are in array-replicated subtrees but are not directly in

Table 3.6
Cost Values for Breadth-First Expansion of RISC II Instruction Cache

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	2	208	208
	INSTANCE Relation	1	4	113
	Stack	0	0	0
	Total	3	212	321
Disk Reads	TREE Relation	329	1	1
	INSTANCE Relation	6326	320	113
Disk Writes	TREE Relation	329	329	329
Disk Accesses		6984	650	443
Disk Time @ 30 msec.		210 sec.	20 sec.	13 sec.
Tuples Read	TREE Relation	9854	9854	9854
	INSTANCE Relation	6325	6325	6325
Tuples Composed	TREE Relation	9853	9853	9853
Tuples Processed		26032	26032	26032
CPU Time @ 3 msec/tuple		78 sec.	78 sec.	78 sec.
Total Time		288 sec.	98 sec.	91 sec.

Table 3.7
Cost Values for Depth-First Expansion of RISC II CPU

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	1	1	1
	INSTANCE Relation	11	21	259
	Stack	4	4	4
	Total	16	26	264
Disk Reads	TREE Relation	1	1	1
	INSTANCE Relation	11465	942	259
Disk Writes	TREE Relation	469	469	469
Disk Accesses		11935	1412	729
Disk Time @ 30 msec.		358 sec.	42 sec.	22 sec.
Tuples Read	TREE Relation	0	0	0
	INSTANCE Relation	11464	11464	11464
Tuples Composed	TREE Relation	14049	14049	14049
Tuples Processed		25513	25513	25513
CPU Time @ 3 msec/tuple		77 sec.	77 sec.	77 sec.
Total Time		435 sec.	119 sec.	99 sec.

Table 3.8
Cost Values for Depth-First Expansion of RISC II Instruction Cache

		Minimum Buffer	Bigger Buffer	Infinite Buffer
Buffer Pages	TREE Relation	1	1	1
	INSTANCE Relation	5	7	113
	Stack	2	2	2
	Total	8	10	116
Disk Reads	TREE Relation	1	1	1
	INSTANCE Relation	6326	320	113
Disk Writes	TREE Relation	329	329	329
Disk Accesses		6656	650	443
Disk Time @ 30 msec.		200 sec.	20 sec.	13 sec.
Tuples Read	TREE Relation	0	0	0
	INSTANCE Relation	6325	6325	6325
Tuples Composed	TREE Relation	9853	9853	9853
Tuples Processed		16178	16178	16178
CPU Time @ 3 msec/tuple		49 sec.	49 sec.	49 sec.
Total Time		249 sec.	69 sec.	62 sec.

arrays. An additional factor of two is gained with the unlimited buffering scheme.

(3) The choice of expansion algorithm affects I/O very little. However the depth-first algorithm requires much less buffer space for all except the minimum buffer policies.

(4) The "unlimited" buffer size is quite feasible, if the design trees tested are representative. The largest calculated buffer size is less than 600 kilobytes for breadth-first, 300 for depth-first. Large tuples in the TREE and INSTANCE

relations would change this, but an increase by more than a factor of two or three seems unlikely.

(5) Depth-first expansion requires much less buffer space than breadth-first, as would be expected for such wide, shallow trees. The factor varies between two and twenty.

(6) The depth-first algorithm has consistently lower CPU time estimates because it processes fewer tuples than breadth-first.

These calculations indicate that depth-first expansion is faster and uses less memory space. It should be used in combination with the unlimited buffer policy, because this gives the best performance and still does not require excessive space.

3.8. Append* Implementation Tests

In order to validate the foregoing analysis, and to investigate how transitive closure operations should be implemented in a real system, we added *append** to INGRES and used our implementation to expand real VLSI design trees. In this section we describe a series of tests designed to evaluate the costs and performance effects of several choices in the processing of transitive closure queries. These include breadth-first versus depth-first tree expansion, three kinds of duplicate tuple elimination (see Section 3.5.3), and tree traversal with and without materialization of the result. Although the tests involve expansion of design trees, the same processing techniques are applicable to other kinds of transitive closure queries as well.

First in Section 3.0(2.1 we describe the database schema, the queries used to expand a design tree, and modifications to INGRES. The results are tabulated and discussed in Section 3.8.2, and Section 3.8.3 presents our conclusions.

3.8.1. Description of the Experiment

For the tests we used VLSI data from the two designs described in the preceding section, the RISC II CPU and Instruction Cache. Hierarchical structuring information for each design was stored in an INSTANCE relation of the following form:

INSTANCE (owner, cell, t00, t10, t20, t01, t11, t21).

This relation definition is very similar to the one presented in Section 3.6.1, but the 3×2 transform matrix is represented as six individual numbers *t00* - *t21* because current INGRES does not have matrix data types. The *owner* and *cell* fields contain the identifying numbers of the parent and child cells.

The QUEL statements below served to expand a design and produce a TREE relation containing a tuple for each node in the tree. Again these are similar to the tree-expansion queries given earlier, except that the matrix multiplication required to compose spatial transforms is expressed in QUEL.

```
create TREE (cell, gener, t11, t12, t21, t22, t31, t32)
range of t is TREE
range of i is INSTANCE
append to TREE (cell=1, gener=0, t11=1, t22=1)
append* to TREE (cell = i.cell, gener = t.gener+1,
        t11 = t.t11 * i.t00 + t.t12 * i.t10,
        t12 = t.t11 * i.t01 + t.t12 * i.t11,
        t21 = t.t21 * i.t00 + t.t22 * i.t10,
        t22 = t.t21 * i.t01 + t.t22 * i.t11,
        t31 = t.t31 * i.t00 + t.t32 * i.t10 + i.t20,
        t32 = t.t31 * i.t01 + t.t32 * i.t11 + i.t21)
where t.cell = i.owner
```

The first *append* statement installs one tuple to represent the root of the tree. Then the *append** repeatedly reads tuples from the TREE relation and composes new tuples corresponding to their children by referring to the INSTANCE relation. Each TREE tuple includes the absolute location of its cell, which insures that no duplicate tuples are produced. The *gener* (generation) field records the distance of each node from the root; when a new tuple is generated its *gener* value is one greater than its parent's.

A second set of queries expanded a design tree in the same way but produced a TREE relation with many duplicate tuples. This version stored no information in TREE to distinguish between instances of the same circuit cell:

```
create TREE (cell)
range of t is TREE
range of i is INSTANCE
append to TREE (cell=1)
append* to TREE (cell = i.cell)
        where t.cell = i.owner
```

Standard INGRES normally processes queries in two steps in order to make recovery easier after interruptions. During the first phase the source relations are read and updates are written to a separate deferred-update file, and during the second phase the updates are applied to the target relation. INGRES can also be made to perform updates directly, without the two-phase protocol. Both methods were used in our experiments because they interacted with the expansion algorithms; recovery methods were not under consideration.

One set of tests performed tree expansion by using direct update with an essentially unmodified version of INGRES. (The only change consisted of installing counters for the experiment.) The TREE relation was structured as a heap, i.e. tuples were stored one after another in the order in which they were inserted. INGRES performed an ordinary *append* query instead of the *append**, reading parent tuples from the relation and added new child tuples immediately to the end, where they would be used as parents later during processing of the same query. In this way a single *append* was made to process one level after another and expand an entire tree in a breadth-first manner. Eliminating duplicate tuples during processing is not practical with this method.

Our second version of INGRES repeated *append* queries automatically, using a modified form of deferred update, to make them perform like *append**. During the first phase of each iteration INGRES scanned the source relations (TREE and INSTANCE in our tests), composed new tuples, and stored them in a deferred-update file. During the second phase it appended each new tuple to the target relation (TREE) and also to a special temporary relation. During the next iteration it used the temporary as a source relation in place of TREE. In our tests, for example, INGRES first read the single root tuple in TREE and composed tuples corresponding to children of the root, writing them to a file. Then it read the file, adding the new tuples to TREE and to TEMP. During the second iteration it read tuples from TEMP and composed tuples for nodes in the third level of the tree, again saving them in a file. Then it emptied TEMP and added the second batch of new tuples to TREE and to TEMP. Processing continued in this way until one iteration produced no new tuples.

We implemented several variations on this version of INGRES. The first used standard INGRES's handling of hash-structured relations to eliminate duplicate tuples in TREE. When INGRES prepares to append a new tuple to a hashed relation, it checks all tuples in the appropriate bucket and suppresses adding the new one if it is a duplicate. Our implementation first attempted to add new tuples to a hashed TREE relation, and if they were duplicates it did not add them either to TREE or to TEMP. Sorting TEMP between iterations provided a second, independent kind of duplicate elimination which removed duplicates only within one level of the tree. A third variation processed an *append** without actually producing a result relation (TREE in our tests). This was accomplished by omitting appends to the result relation but retaining those to TEMP.

Our third modified version of INGRES performed depth-first tree expansion recursively as described in Section 3.6.5. A mutually recursive set of routines within INGRES was invoked to find children of the root node, and they called themselves with each new tuple. Preliminaries such as parsing the query were performed only once. Global duplicate tuple elimination was done as before by hashing the result relation. Duplicate elimination within one level was impractical, but it was possible to eliminate the third kind of duplicates, those along a path from the root to a new tuple. This was accomplished by keeping all tuples for recursive invocations of the routines on a linked list, a sort of "ancestor list". Each new tuple was checked against all others in the list as soon as it was composed, and if a duplicate was found the next recursive call was suppressed and the new tuple was not added to the result.

We did not implement the buffer management methods of the preceding section; our modified versions of INGRES used the same LRU buffer management as standard INGRES, and the same small buffer (15 pages). CPU and elapsed time were measured by means of the UNIX *time* command [42], and each version of INGRES had counters to record the number of logical disk reads and writes. The tests were run on a very lightly loaded VAX 11/780 computer; essentially all the resources of the machine were available.

3.8.2. Test Results

Sixty-four tests were run in all, and the complete results are given in Appendix B. Tables 3.9-3.15 display data from selected pairs of tests in order to show the effects of each processing option. The test numbers shown in each table can be used to look up more complete information about individual tests by referring to the Appendix.

Table 3.9 compares breadth-first to depth-first tree expansion, and shows that depth-first expansion was faster in terms of I/O counts, CPU time, and elapsed time. The main reason was probably that the recursive depth-first expansion program saved TREE tuples it would need later on the stack, whereas breadth-first INGRES put them in relations and incurred a much higher cost for storing and retrieving them. In addition, the design trees we used were much wider than they were tall, so breadth-first INGRES had to store many more tuples for one level of the tree than the depth-first version did for a vertical slice.

The tests shown in Table 3.9 tended to use much more CPU than I/O time, and in fact this was true of all 64 tests. INGRES's use of the CPU could probably be made more efficient, for instance by compiling rather than interpreting queries [93], but even a large improvement would be overwhelmed in a multi-user system with many disk drives connected to a single CPU. We expect that applications similar to our tests (i.e. those in which a general-purpose database system answers queries involving many tuples sharing pages) will continue to be CPU bound. This may not be true for data managers like SQUID [49] that are especially designed for CAD and which use the CPU more efficiently, but still must fetch data from disk.

Table 3.10 shows the cost of using a hash structured result relation to eliminate duplicate tuples during breadth-first tree expansion. This caused pages of the TREE

Table 3.9
Comparison of Breadth-First and Depth-First Expansion

	Test No.	Algorithm	Disk Reads	Disk Writes	Disk Sec. @ 30 msec.	CPU Sec.	Elapsed Sec.
RISC	1	Breadth-first	1247	1831	92	301	333
CPU	5	Depth-first	805	907	51	257	281
RISC	17	Breadth-first	2839	1279	124	267	301
Cache	21	Depth-first	2517	635	95	239	266

relation to be accessed randomly instead of sequentially, and also meant that the relation was spread over more pages. When there were no duplicates to be found I/O cost was increased by a factor of 2 to 7 and CPU cost was approximately doubled. When duplicates were present they were detected during the second phase of each iteration when updates were being applied to the TREE relation, and were not added either to TREE or to TEMP. This prevented useless expansion of identical subtrees, and decreased I/O, CPU and elapsed times by large amounts.

Table 3.10
Global Duplicate Tuple Elimination During Breadth-First Expansion

	Test No.	Global Dup Elim	Tuples in Result	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	1	No	14050	1247	1831	301	333
No Dups	3	Yes	14050	5560	4003	568	733
Cache	17	No	9854	2839	1279	267	301
No Dups	19	Yes	9854	5149	2294	463	575
CPU	33	No	14050	878	353	161	179
Many Dups	35	Yes	259	438	70	27	43
Cache	49	No	9854	2581	245	173	188
Many Dups	51	Yes	113	383	35	27	37

Table 3.11 summarizes the more complicated results for depth-first expansion. Duplicates could be detected during tree traversal only if updates were performed immediately; if they were deferred until a second phase duplicates were not discovered until after the entire tree had been traversed, and as a result no time savings were realized. Even this late detection eliminated many disk writes, because each suppressed update usually meant that a relation page did not have to be written out to disk. This saving almost compensated for the extra I/O due to random reads for the hashed relation structure (e.g. in tests 37-39, 53-55).

With direct update duplicates could be detected during tree traversal and the improvement was comparable to that of breadth-first expansion. This was in spite of another curious phenomenon appearing in the same tests, namely that direct update drastically increased the number of disk writes (tests 39-40, 55-56). During direct-update processing the INGRES buffer was used for INSTANCE pages, TREE pages being read, and other TREE pages being written. The small size of the buffer (15 pages) meant that modified TREE pages often had to be written out and fetched again later. With updates deferred, relation pages were not modified during the first phase and never had to be written to disk. During the second phase when updates were applied, all ten TREE relation pages could remain in the buffer.

Table 3.12 shows the results of eliminating duplicates within each level of a tree during breadth-first expansion by sorting the temporary relation between iterations. The cost was less than for global duplicate elimination by hashing, and increased I/O, CPU, and elapsed time by less than a factor of two when there were no duplicates. When duplicates were present large savings were obtained, nearly as great as with

Table 3.11
Global Duplicate Tuple Elimination During Depth-First Expansion

	Test No.	Global Dup Elim	Direct Update	Tuples in Result	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	5	No	No	14050	805	907	257	281
No	7	Yes	No	14050	4667	3380	523	678
Dups	8	Yes	Yes	14050	7564	4149	532	747
Cache	21	No	No	9854	2517	635	239	266
No	23	Yes	No	9854	3874	1658	428	508
Dups	24	Yes	Yes	9854	3936	1659	432	531
CPU	37	No	No	14050	805	167	123	138
Many	39	Yes	No	259	815	10	132	143
Dups	40	Yes	Yes	259	493	109	20	33
Cache	53	No	No	9854	2517	117	149	168
Many	55	Yes	No	113	2527	10	152	165
Dups	56	Yes	Yes	113	394	51	23	31

global duplicate elimination. This was because most duplication in our design trees was due to repeated nodes or subtrees at the same level in the tree.

Table 3.13 indicates that checking for duplicates on the "ancestor list" between a new node and the root of the tree cost very little. I/O was not affected at all, and the

Table 3.12
Duplicate Tuple Elimination Within One Tree Level
During Breadth-First Expansion

	Test No.	Dup Elim Within Level	Tuples in Result	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	1	No	14050	1247	1831	301	333
No Dups	4	Yes	14050	1239	2745	491	551
Cache	17	No	9854	2839	1279	267	301
No Dups	20	Yes	9854	2843	1919	389	435
CPU	33	No	14050	878	353	161	179
Many Dups	36	Yes	2881	410	115	33	57
Cache	49	No	9854	2581	245	173	188
Many Dups	52	Yes	3763	359	107	34	50

extra processing cost was negligible because the list was in memory and never contained more than 9 tuples. For very long derivation chains this method might not work so well, because its cost increases linearly with the length of the chains. Unfortunately this kind of duplicate elimination offered no possibility of improving processing efficiency with our design data, which contained no duplicates along a vertical path through the tree. In our schema a duplicate of this kind would represent the use of a circuit cell as a part within itself, which does not make sense.

Table 3.13
Duplicate Tuple Elimination on Path to Root
During Depth-First Expansion

	Test Number	Dup Elim on Path to Root	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	5	No	805	907	257	281
	9	Yes	805	907	259	283
Cache	21	No	2517	635	239	266
	25	Yes	2517	635	242	269

Table 3.14 compares the cost of traversing a design tree without materializing it versus building a heap-structured relation. I/O is decreased, especially for depth-first expansion, but the saving in CPU and elapsed time is small. The main advantage of not materializing a tree would be to save space, in the case of extremely large trees.

Table 3.15 shows the performance effects of using direct or deferred update in cases where the two were functionally equivalent. The choice usually made little difference in the efficiency of tree expansion. Direct update decreased disk writes by about a factor of two for breadth-first expansion, because there was no update file to be written and then read, and CPU and elapsed times were slightly smaller. In the case of depth-first expansion with global duplicate elimination, direct update changed the way the query was processed, and this had a strong effect on processing time, as noted above.

Table 3.14
Cost of Materializing the Tree Relation

	Test Number	Algorithm	Tuples in Result	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	1	Breadth-first	14050	1247	1831	301	333
	13	Breadth-first	0	1238	915	280	308
	9	Depth-first	14050	805	907	259	283
	16	Depth-first	0	804	0	231	241
Cache	17	Breadth-first	9854	2839	1279	267	301
	29	Breadth-first	0	2836	641	258	286
	25	Depth-first	9854	2517	635	242	269
	32	Depth-first	0	2516	0	221	237

Table 3.15
Direct vs. Deferred Update

	Test No.	Algorithm	Updates	Disk Reads	Disk Writes	CPU Sec.	Elapsed Sec.
CPU	1	Breadth-first	Deferred	1247	1831	301	333
	2	Breadth-first	Direct	1227	909	314	278
	5	Depth-first	Deferred	805	907	257	281
	6	Depth-first	Direct	814	907	250	264
Cache	17	Breadth-first	Deferred	2839	1279	267	301
	18	Breadth-first	Direct	2830	636	265	301
	21	Depth-first	Deferred	2517	635	239	266
	22	Depth-first	Direct	2521	635	238	256

3.8.3. Conclusions

Depth-first tree expansion appears to be better than breadth-first, at least for design trees, because it makes use of main memory to store tuples temporarily, instead of putting them in database relations or files. It also requires less buffer space when the trees are wide rather than tall.

Depth-first processing also facilitates elimination of duplicate tuples along a chain of derivation, which is required for correct processing of some queries. In applications where very long derivation chains are expected it may be worthwhile to devise a more sophisticated method than the ancestor list scan used in our tests.

Duplicate tuple elimination within each level of a tree must be abandoned if depth-first expansion is to be used. Global duplicate elimination should be optional, to be used to speed processing in cases where many duplicates are expected. Our hashing scheme worked well for the examples we tried, but with this method it is necessary to guess a reasonable number of hash buckets in advance. For large or unpredictable applications the result relation could be structured as a B-tree; this would probably increase the amount of I/O, since two or three disk reads may be required to descend a B-tree, but the structure adapts itself to any number of tuples. Extendible hashing [22] is similarly adaptive, again at the cost of extra I/O, since it requires an extra level of indirection.

Materialization of the result may be avoided if global duplicate elimination is not required. This can save considerable space, and is especially useful if the result of a transitive closure operation is passed to another query, as outlined in Section 3.2.

Our conclusion is that *append** is best implemented using a depth-first algorithm. Analysis and tests both confirm that this is faster and uses less space. In addition, duplicate tuple elimination of the path-to-root type is easy to build into a depth-first implementation and does not degrade performance appreciably. Other kinds of duplicate elimination may or may not be beneficial depending on the application and should be optional. Sophisticated use of large main-memory buffers does not appear to be very important for design-tree expansion; our analytic estimates and our tests strongly indicate that a database system will be CPU bound in such applications.

CHAPTER 4

A Dynamic Index Structure for Spatial Searching

VLSI layout rectangles cover overlapping areas in two dimensions and are not well represented by point locations. In this they are like many kinds of spatial data occurring in computer aided design (CAD) and other applications such as geographic databases. For example, map objects like counties, census tracts etc. also occupy regions of non-zero size in two dimensions. A common operation on spatial data is a search for all objects in an area, for example the test described in Chapter 2 in which INGRES retrieved VLSI layout geometry falling within a small "window" in the middle of a circuit cell. For this kind of search it is important to be able to retrieve objects efficiently according to their spatial locations.

An index based on objects' spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using one-dimensional ordering of key values, such as B-trees and ISAM indexes, do not work because the search space is multi-dimensional.

A number of structures have been proposed for handling multi-dimensional point data, and a survey of methods can be found in [10]. Cell methods [9,32,100] are not good for dynamic structures because the cell boundaries must be decided in advance. Quad trees [25] and k-d trees [8] do not take paging of secondary memory into account. K-D-B trees [84] are designed for paged memory but are useful only for point data. The use of index intervals has been suggested in [97], but this method cannot be used in multiple dimensions. Corner stitching [77] is an example of a structure for two-dimensional spatial searching suitable for data objects of non-zero size, but it assumes homogeneous primary memory and is not efficient for random searches in very large collections of data. Grid files [35] handle non-point data by mapping each object to a point in a higher-dimensional space. In this chapter we describe an alternative structure called an R-tree which represents data objects by intervals in several dimensions.

Section 4.2 outlines the structure of an R-tree and Section 4.3 gives algorithms for searching, inserting, deleting, and updating. The problem of splitting over-full tree nodes, which is part of the insertion process, is considered in Section 4.4. Results of R-tree index performance tests are presented in Section 4.5, and Section 4.6 we outline how to implement R-trees in INGRES [34].

4.1. R-Tree Index Structure

An R-tree is a height-balanced tree similar to a B-tree [5,18] with index records in its leaf nodes containing pointers to data objects. Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form

$$(I,\ tuple-identifier)$$

where *tuple−identifier* refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed:

$$I = (I_0, I_1, \ldots, I_{n-1})$$

Here n is the number of dimensions and I_i is a closed bounded interval $[a,b]$ describing the extent of the object along dimension i. Alternatively I_i may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely. Non-leaf nodes contain entries of the form

$$(I,\ child-pointer)$$

where *child−pointer* is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries.

Let M be the maximum number of entries that will fit in one node and let $m \leq \frac{M}{2}$ be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

(1) Every leaf node contains between m and M index records unless it is the root.

(2) For each index record $(I,\ tuple-identifier)$ in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.

(3) Every non-leaf node has between m and M children unless it is the root.

(4) For each entry (I, *child–pointer*) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.

(5) The root node has at least two children unless it is a leaf.

(6) All leaves appear on the same level.

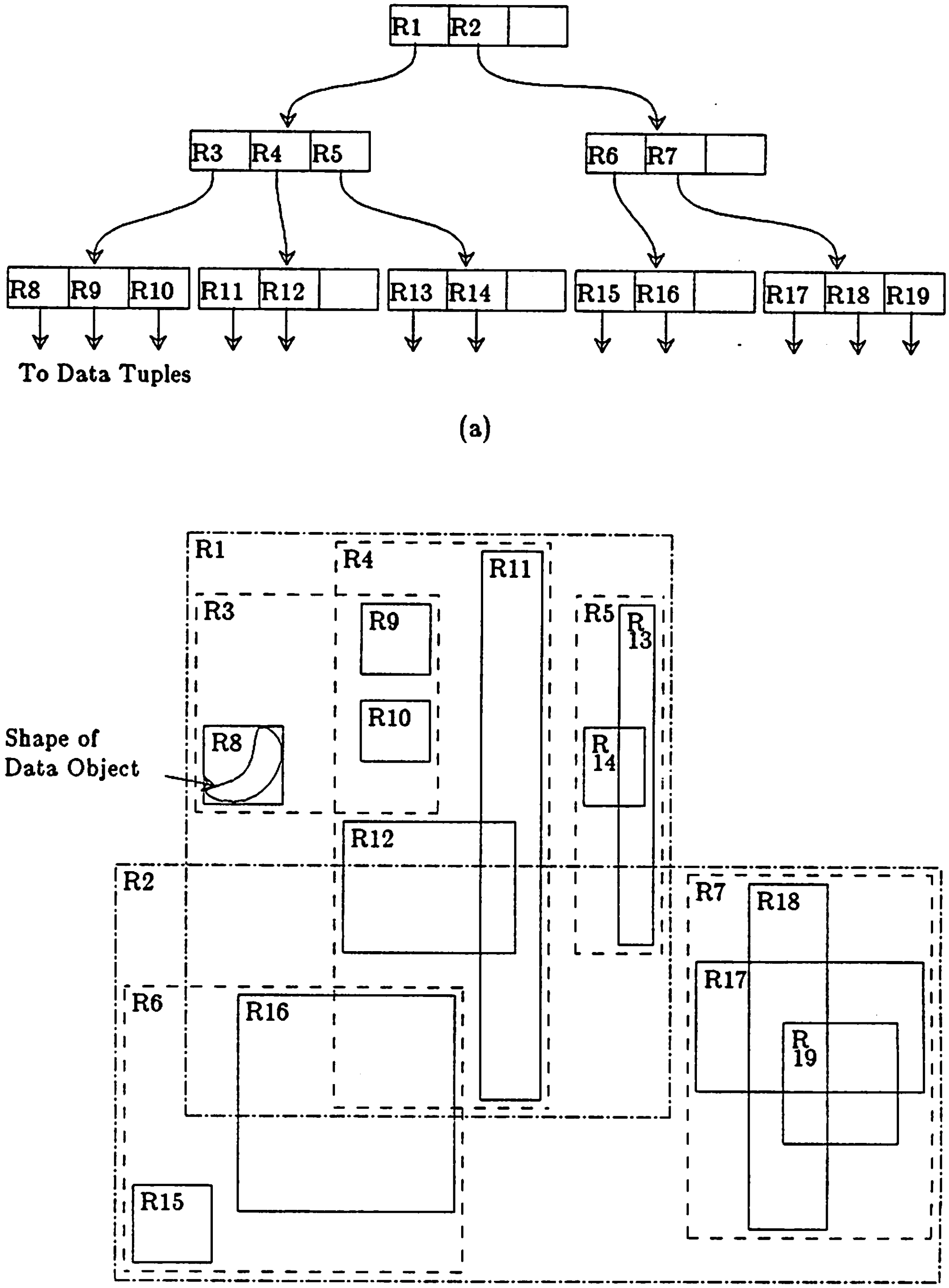
Figure 4.1a and 4.1b show the structure of an R-tree and illustrate the containment and overlapping relationships that can exist between its rectangles.

The height of an R-tree containing N index records is at most $\left\lceil \log_m N \right\rceil - 1$, because the branching factor of each node is at least m. The maximum number of nodes is $\left\lceil \frac{N}{m} \right\rceil + \left\lceil \frac{N}{m^2} \right\rceil + \cdots + 1$. Worst-case space utilization for all nodes except the root is $\frac{m}{M}$. Nodes will tend to have more than m entries, and this will decrease tree height and improve space utilization. If nodes have more than 3 or 4 entries the tree is very wide, and almost all the space is used for leaf nodes containing index records. The parameter m can be varied as part of performance tuning, and different values are tested experimentally in Section 4.5.

4.2. Searching and Updating

4.2.1. Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. However when it visits a non-leaf node it may find that any number of


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
To Data Tuples
Shape of
Data Object

(a)

(b)

Figure 4.1

subtrees need to be searched, hence it is not possible to guarantee good worst-case performance. Nevertheless with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area.

In the following we denote the rectangle part of an index entry E by $E.I$, and the *tuple−identifier* or *child−pointer* part by $E.p$.

Algorithm **Search.** Given an R-tree whose root node is T, find all index records whose rectangles overlap a search rectangle S.

S1. [Search subtrees.] If T is not a leaf, check each entry E to determine whether $E.I$ overlaps S. For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by $E.p$.

S2. [Search leaf node.] If T is a leaf, check all entries E to determine whether $E.I$ overlaps S. If so, E is a qualifying record.

4.2.2. Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

Algorithm **Insert**. Insert a new index entry E into an R-tree.

I1. [Find position for new record.] Invoke **ChooseLeaf** to select a leaf node L in which to place E.

I2. [Add record to leaf node.] If L has room for another entry, install E. Otherwise invoke **SplitNode** to obtain L and LL containing E and all the old entries of L.

I3. [Propagate changes upward.] Invoke **AdjustTree** on L, also passing LL if a split was performed.

I4. [Grow tree taller.] If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Algorithm **ChooseLeaf**. Select a leaf node in which to place a new index entry E.

CL1. [Initialize.] Set N to be the root node.

CL2. [Leaf check.] If N is a leaf, return N.

CL3. [Choose subtree.] If N is not a leaf, let F be the entry in N whose rectangle $F.I$ needs least enlargement to include $E.I$. Resolve ties by choosing the entry with the rectangle of smallest area.

CL4. [Descend until a leaf is reached.] Set N to be the child node pointed to by $F.p$ and repeat from CL2.

Algorithm **AdjustTree**. Ascend from a leaf node L to the root, adjusting covering rectangles and propagating node splits as necessary.

AT1. [Initialize.] Set N=L. If L was split previously, set NN to be the resulting second node.

AT2. [Check if done.] If N is the root, stop.

AT3. [Adjust covering rectangle in parent entry.] Let P be the parent node of N, and let E_N be N's entry in P. Adjust $E_N.I$ so that it tightly encloses all entry rectangles in N.

AT4. [Propagate node split upward.] If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.p$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN. Add E_{NN} to P if there is room. Otherwise, invoke **SplitNode** to produce P and PP containing E_{NN} and all P's old entries.

AT5. [Move up to next level.] Set N=P and set NN=PP if a split occurred. Repeat from AT2.

Algorithm **SplitNode** is described in Section 4.4.

4.2.3. Deletion

Algorithm **Delete**. Remove index record E from an R-tree.

D1. [Find node containing record.] Invoke **FindLeaf** to locate the leaf node L containing E. Stop if the record was not found.

D2. [Delete record.] Remove E from L.

D3. [Propagate changes.] Invoke **CondenseTree**, passing L.

D4. [Shorten tree.] If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm **FindLeaf**. Given an R-tree whose root node is T, find the leaf node containing the index entry E.

FL1. [Search subtrees.] If T is not a leaf, check each entry F in T to determine if $F.I$ overlaps $E.I$. For each such entry invoke **FindLeaf** on the tree whose root is pointed to by $F.p$ until E is found or all entries have been checked.

FL2. [Search leaf node for record.] If T is a leaf, check each entry to see if it matches E. If E is found return T.

Algorithm **CondenseTree**. Given a leaf node L from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them smaller if possible.

CT1. [Initialize.] Set $N=L$. Set Q, the set of eliminated nodes, to be empty.

CT2. [Find parent entry.] If N is the root, go to CT6. Otherwise let P be the parent of N, and let E_N be N's entry in P.

CT3. [Eliminate under-full node.] If N has fewer than m entries, delete E_N from P and add N to set Q.

CT4. [Adjust covering rectangle.] If N has not been eliminated, adjust $E_N.I$ to tightly contain all entries in N.

CT5. [Move up one level in tree.] Set $N=P$ and repeat from CT2.

CT6. [Re-insert orphaned entries.] Re-insert all entries of nodes in set Q. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in

Algorithm **Insert**, but entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

The procedure outlined above for disposing of under-full nodes differs from the corresponding operation on a B-tree, in which two or more adjacent nodes are merged. A B-tree-like approach is possible for R-trees, although there is no adjacency in the B-tree sense: an under-full node can be merged with whichever sibling will have its area increased least, or the orphaned entries can be distributed among sibling nodes. Either method can cause nodes to be split. We chose re-insertion instead for two reasons: first, it accomplishes the same thing and is easier to implement because the **Insert** routine can be used. Efficiency should be comparable because pages needed during re-insertion usually will be the same ones visited during the preceding search and will already be in memory. The second reason is that re-insertion incrementally refines the spatial structure of the tree, and prevents gradual deterioration that might occur if each entry were located permanently under the same parent node.

4.2.4. Updates and Other Operations

If a data tuple is updated so that its covering rectangle is changed, its index record must be deleted, updated, and then re-inserted, so that it will find its way to the right place in the tree.

Other kinds of searches besides the one described above may be useful, for example to find all data objects completely contained in a search area, or all objects that contain a search area. These operations can be implemented by straightforward variations on the algorithm given. A search for a specific entry whose identity is known beforehand is required by the deletion algorithm and is implemented by Algorithm **FindLeaf**. Variants of range deletion, in which index entries for all data objects in a particular area are removed, are also well supported by R-trees.

4.3. Node Splitting

In order to add a new entry to a full node containing M entries, it is necessary to divide the collection of $M+1$ entries between two nodes. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since the decision whether to visit a node depends on whether its covering rectangle overlaps the search area, the total area of the two covering rectangles after a split should be minimized. Figure 4.2 illustrates this point. The area of the covering rectangles in the "bad split" case is much larger than in the "good split" case.

The same criterion was used in procedure **ChooseLeaf** to decide where to insert a new index entry: at each level in the tree, the subtree was chosen whose covering rectangle would have to be enlarged least.

We now turn to algorithms for partitioning the set of $M+1$ entries into two groups, one for each new node.

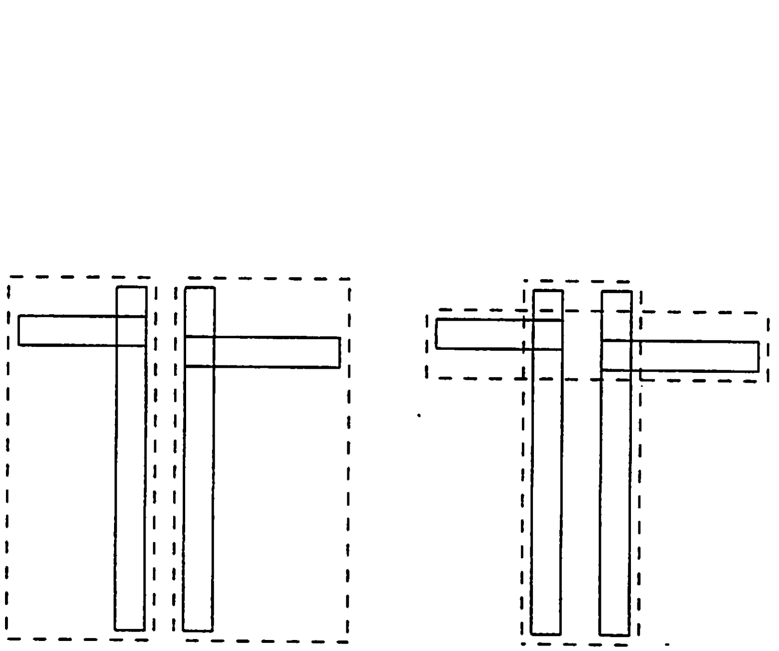


Figure 4.2

4.3.1. Exhaustive Algorithm

The most straightforward way to find the minimum area node split is to generate all possible groupings and choose the best. However, the number of possibilities is approximately 2^{M-1} and a reasonable value of M is 50[*], so the number of possible splits is very large. We implemented a modified form of the exhaustive algorithm to use as a standard for comparison with other algorithms, but it was too slow to use with large page sizes.

4.3.2. A Quadratic-Cost Algorithm

This algorithm attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible. The cost is quadratic in M and linear in the number of dimensions. In other words, if the number of entries in a node is doubled

[*] A two dimensional rectangle can be represented by four numbers of four bytes each. If a pointer also takes four bytes, each entry requires 20 bytes. A page of 1024 bytes will hold about 50 entries.

the split algorithm becomes four times as expensive. With the same number of entries in a node, four-dimensional data makes the algorithm twice as expensive as two-dimensional. The algorithm picks two of the M+1 entries to be the first elements of the two new groups by choosing the pair that would waste the most area if both were put in the same group, i.e. the area of a rectangle covering both entries, minus the areas of the entries themselves, would be greatest. The remaining entries are then assigned to groups one at a time. At each step the area expansion required to add each remaining entry to each group is calculated, and the entry assigned is the one showing the greatest difference between the two groups.

Algorithm **Quadratic Split**. Divide a set of M+1 index entries into two groups.

QS1. [Pick first entry for each group.] Apply Algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.

QS2. [Check if done.] If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.

QS3. [Select entry to assign.] Invoke Algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

Algorithm **PickSeeds.** Select two entries to be the first elements of the groups.

PS1. [Calculate inefficiency of grouping entries together.] For each pair of entries E_1 and E_2, compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate $d=$ area(J) - area($E_1.I$) - area($E_2.I$).

PS2. [Choose the most wasteful pair.] Choose the pair with the largest d.

Algorithm **PickNext.** Select one remaining entry for classification in a group.

PN1. [Determine cost of putting each entry in each group.] For each entry E not yet in a group, calculate $d_1=$ the area increase required in the covering rectangle of Group 1 to include $E.I$. Calculate d_2 similarly for Group 2.

PN2. [Find entry with greatest preference for one group.] Choose any entry with the maximum difference between d_1 and d_2.

4.3.3. A Linear-Cost Algorithm

This algorithm is linear in M and in the number of dimensions. **Linear Split** is identical to **Quadratic Split** but uses a different version of **PickSeeds. PickNext** simply chooses any of the remaining entries.

Algorithm **LinearPickSeeds.** Select two entries to be the first elements of the groups.

LPS1. [Find extreme rectangles along all dimensions.] Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.

LPS2. [Adjust for shape of the rectangle cluster.] Normalize the separations by dividing by the width of the entire set along the corresponding dimension.

LPS3. [Select the most extreme pair.] Choose the pair with the greatest normalized separation along any dimension.

4.4. Performance Tests

We implemented R-trees in C under Unix on a Vax 11/780 computer, and used our implementation in a series of performance tests whose purpose was to verify the practicality of the structure, to choose values for M and m, and to evaluate different node-splitting algorithms. This section presents the results.

Five page sizes were tested, corresponding to different values of M:

Bytes per Page	Max Entries per Page (M)
128	6
256	12
512	25
1024	50
2048	102

Values tested for m, the minimum number of entries in a node, were $M/2$, $M/3$, and 2. The three node split algorithms described earlier were implemented in different versions of the program. All our tests used two-dimensional data, although the structure and algorithms work for any number of dimensions.

During the first part of each test run the program read geometry data from files and constructed an index tree, beginning with an empty tree and calling *Insert* with each new index record. Insert performance was measured for the last 10% of the records, when the tree was nearly its final size. During the second phase the program called the function *Search* with search rectangles made up using random numbers. 100 searches were performed per test run, each retrieving about 5% of the data. Finally the program read the input files a second time and called the function *Delete* to remove the index record for every tenth data item, so that measurements were taken for scattered deletion of 10% of the index records. The tests were done using Very Large Scale Integrated circuit (VLSI) layout data from the RISC-II computer chip [44]. The circuit cell CENTRAL, containing 1057 rectangles, was used in the tests and is shown in Figure 4.3.

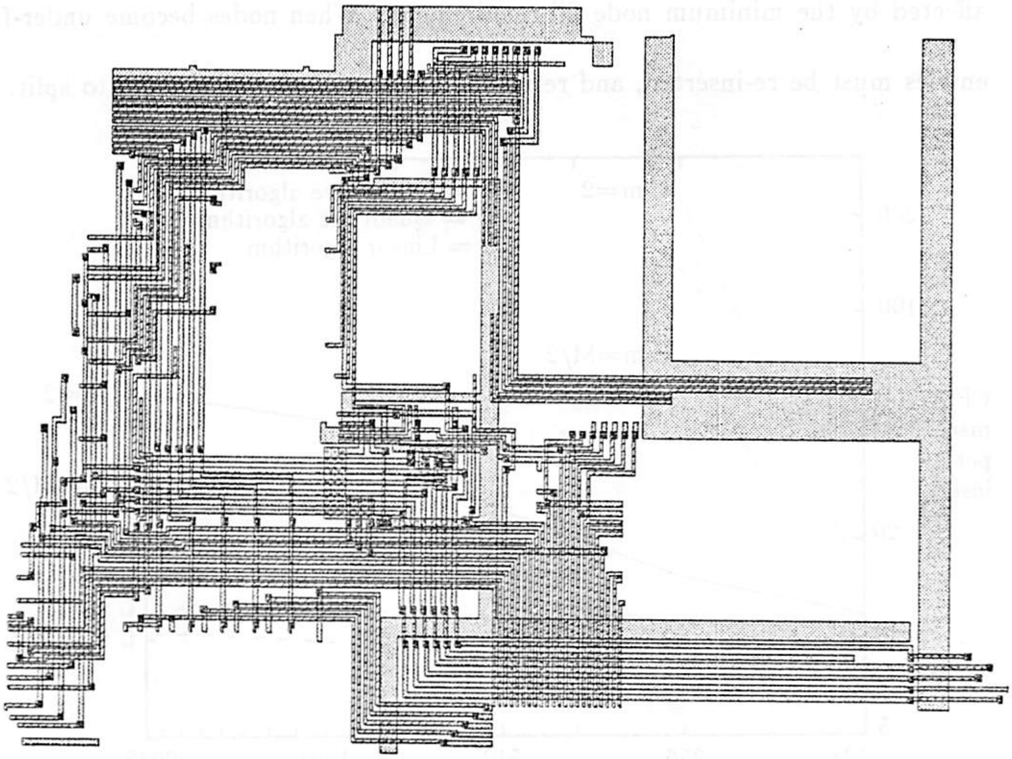


Figure 4.3
Circuit cell CENTRAL (1057 rectangles).

Figure 4.4 shows the cost in CPU time for inserting the last 10% of the records as a function of page size. The exhaustive algorithm, whose cost increases exponentially with page size, is seen to be very slow for larger page sizes. The linear algorithm is fastest, as expected. With this algorithm CPU time hardly increased with page size at all, which suggests that node splitting was responsible for only a small part of the cost of inserting records. The decreased cost of the quadratic algorithm with a stricter node fill requirement reflects the fact that when one group becomes too full, it simply puts all remaining elements in the other group, at linear cost.

The cost of deleting an item from the index, shown in Figure 4.5, is strongly affected by the minimum node fill requirement. When nodes become under-full, their entries must be re-inserted, and re-insertion sometimes causes nodes to split. Stricter

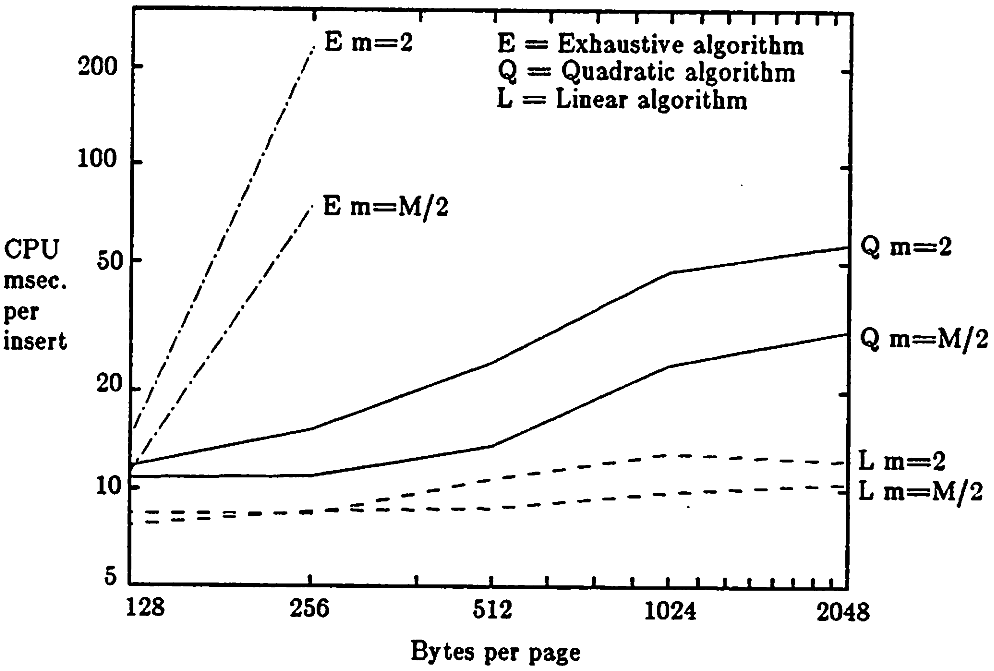


Figure 4.4
CPU cost of inserting records.

fill requirements cause nodes to become under-full more often, and with more entries. Furthermore, splits are more frequent because nodes tend to be fuller. The curves are rough because node eliminations occur randomly and infrequently; there were too few in our tests to smooth out the variations.

Figures 4.6 and 4.7 show that the search performance of the index is very insensitive to the use of different node split algorithms and fill requirements. The exhaustive algorithm produces a slightly better index structure, resulting in fewer pages touched and less CPU cost, but most combinations of algorithm and fill requirement come within 10% of the best. All algorithms provide reasonable performance.

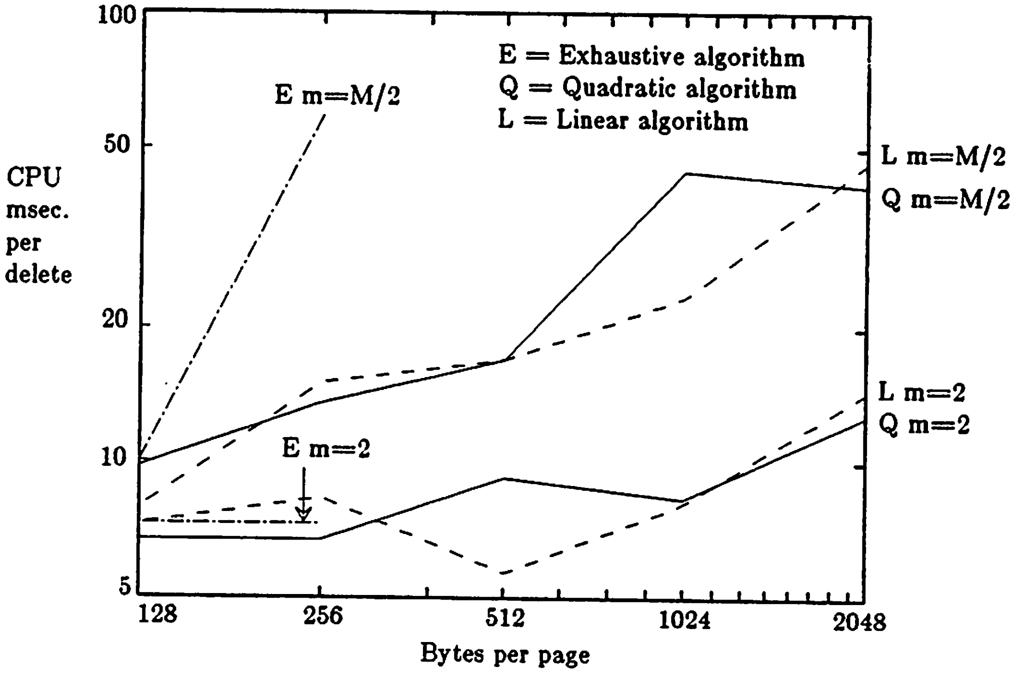


Figure 4.5
CPU cost of deleting records.

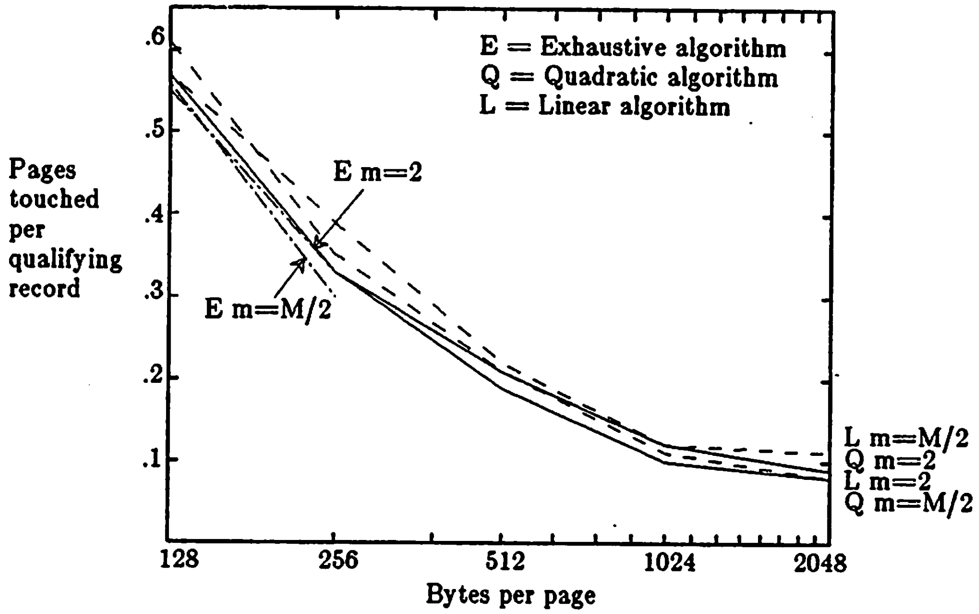


Figure 4.6
Search performance: Pages touched.

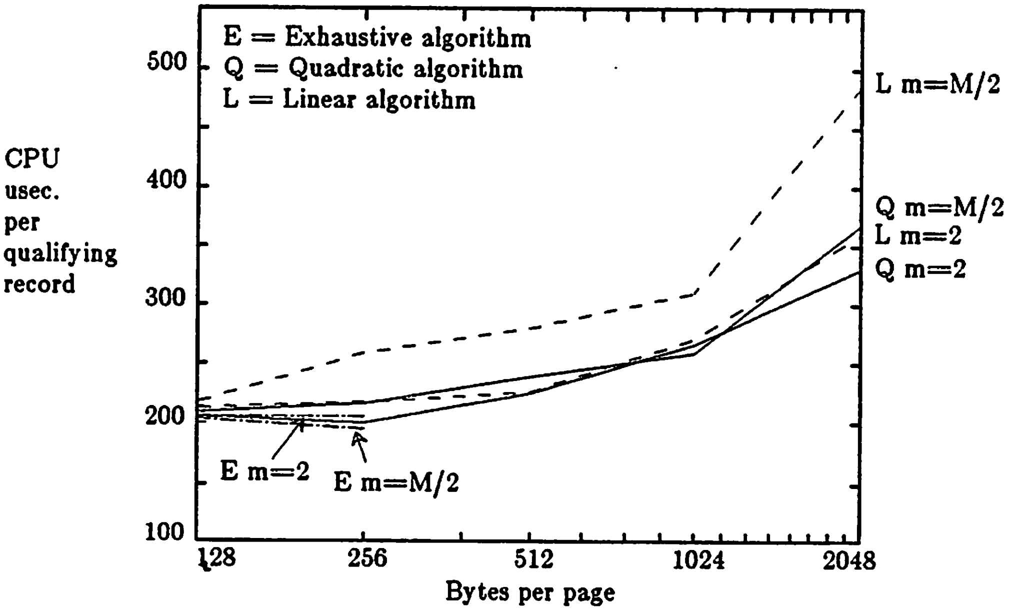


Figure 4.7
Search performance: CPU cost.

Figure 4.8 shows the storage space occupied by the index tree as a function of algorithm, fill criterion and page size. Generally the results bear out our expectation that stricter node fill criteria produce smaller indexes. The least dense index consumes about 50% more space than the most dense, but all results for 1/2-full and 1/3-full (not shown) are within 15% of each other.

A second series of tests measured R-tree performance as a function of the amount of data in the index. The same sequence of test operations as before was run on samples containing 1057, 2238, 3295, and 4559 rectangles. The first sample contained layout data from the circuit cell CENTRAL used earlier, and the second consisted of layout from a similar but larger cell containing 2238 rectangles. The third sample was made by using both CENTRAL and the larger cell, with the two cells effectively

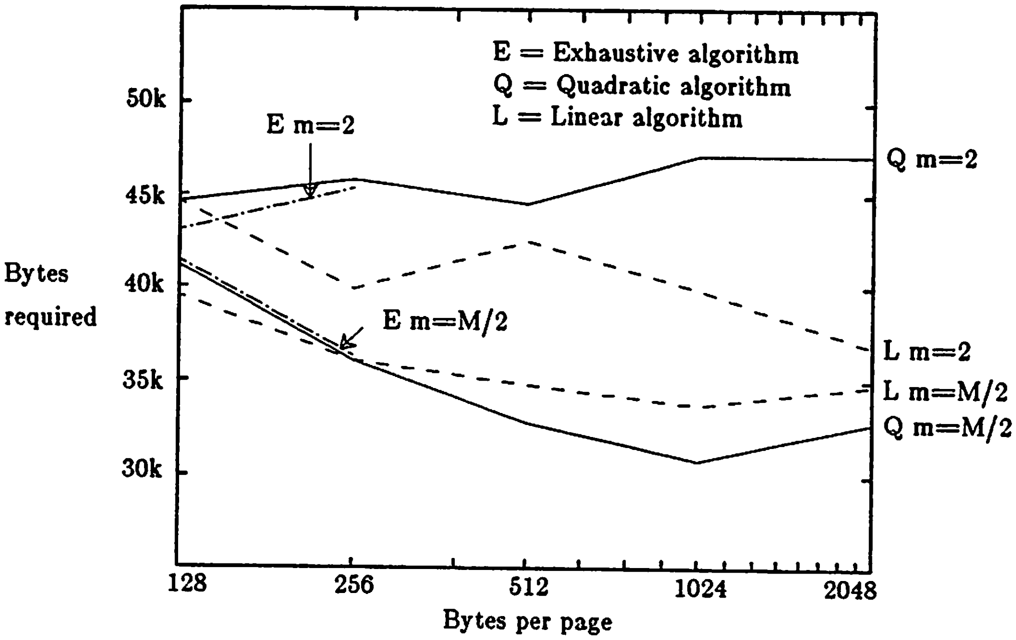


Figure 4.8
Space efficiency.

placed on top of each other. Three cells were combined to make up the last sample. Because the samples were composed in different ways using varying data, performance results do not scale perfectly and some unevenness was to be expected.

Two combinations of split algorithm and node fill requirement were chosen for the tests: the linear algorithm with $m=2$, and the quadratic algorithm with $m=M/3$, both with a page size of 1024 bytes ($M=50$).

Figure 4.9 shows the results of tests to determine how insert and delete performance is affected by tree size. Both test configurations produced trees with two levels for 1057 records and three levels for the other sample sizes. The figure shows that the cost of inserts with the quadratic algorithm is nearly constant except where the tree increases in height. There the curve shows a definite jump because of the increase in the number of levels where a split can occur. The linear algorithm shows no jump, indicating again that linear node splits account for only a small part of the cost of inserts.

No node splits occurred during the deletion tests with the linear configuration, because of the relaxed node fill requirement and the small number of data items. As a result the curve shows only a small jump where the number of tree levels increases. Deletion with the quadratic configuration produced only 1 to 6 node splits, and the resulting curve is very rough. When allowance is made for variations due to the small sample size, the tests show that insert and delete cost is independent of tree width but is affected by tree height, which grows slowly with the number of data items.

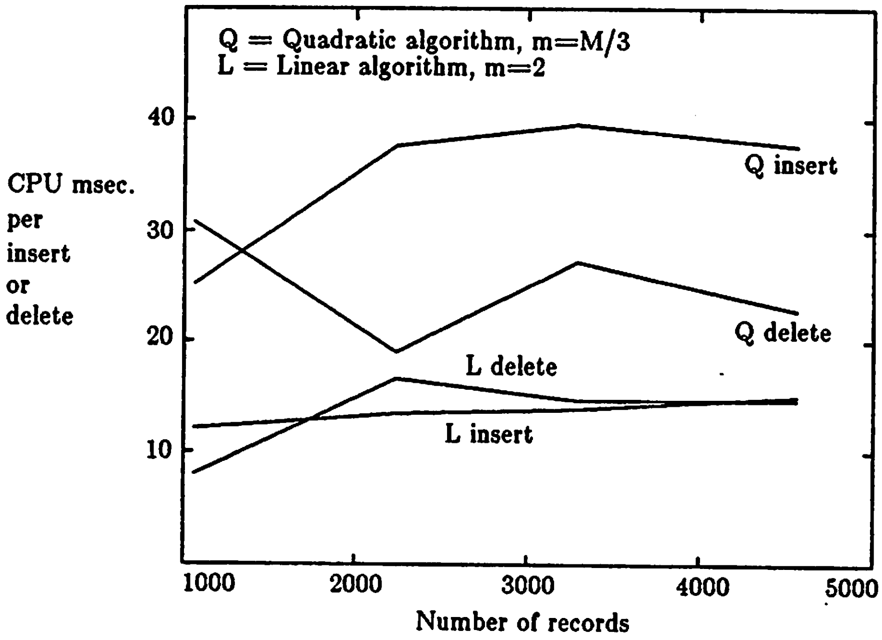


Figure 4.9
CPU cost of inserts and deletes vs. amount of data.

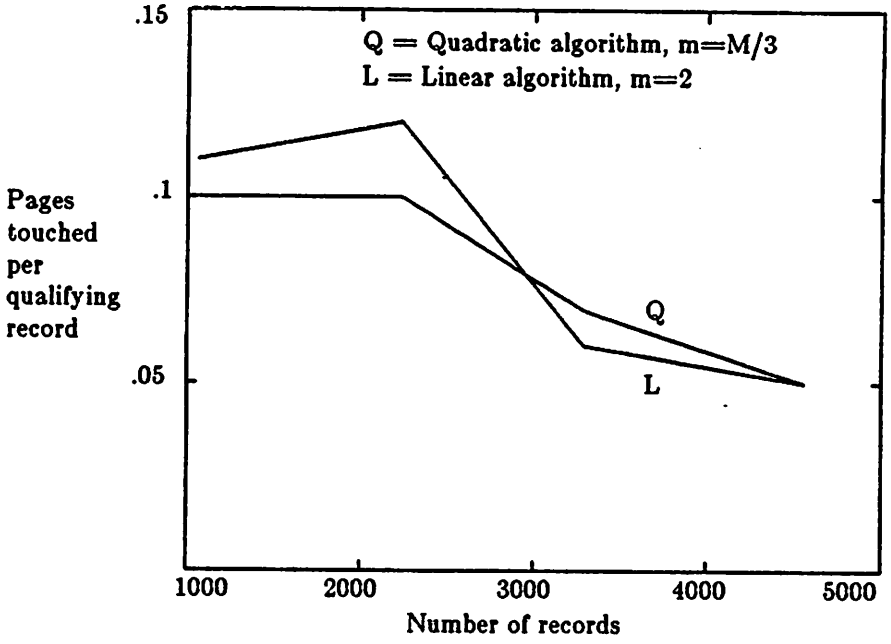


Figure 4.10
Search performance vs. amount of data: Pages touched

Figures 4.10 and 4.11 confirm that the two configurations have nearly the same search performance. Each search retrieved between 3% and 6% of the data. The downward trend of the curves is to be expected, because the cost of processing higher tree nodes becomes less significant as the amount of data retrieved in each search increases. The increase in the number of tree levels kept the cost from dropping between the first and second data points. The low CPU cost per qualifying record, less than 150 microseconds for larger amounts of data, shows that the index is quite effective in narrowing searches to small subtrees.

The straight lines in Figure 4.12 reflect the fact that almost all the space in an R-tree index is used for leaf nodes, whose number varies linearly with the amount of data. For the Linear-2 test configuration the total space occupied by the R-tree was about 40 bytes per data item, compared to 20 bytes per item for the index records

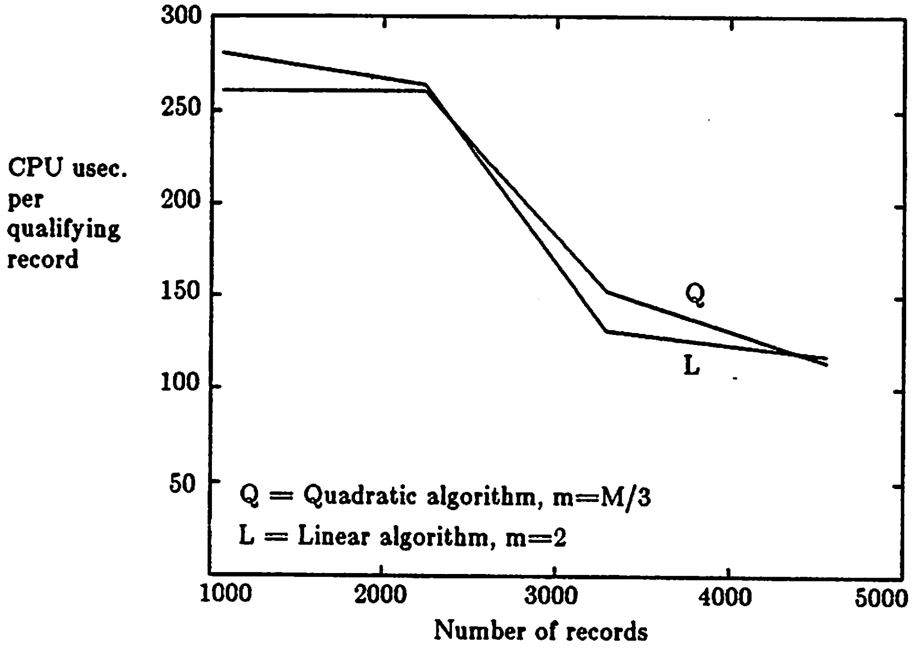


Figure 4.11
Search performance vs. amount of data: CPU cost

alone. The corresponding figure for the Quadratic-1/3 configuration was 33 bytes per item.

4.5. Implementation in INGRES

Current INGRES supports four storage structures for relations: heap, compressed heap, ISAM and hashed. Secondary indexes are implemented as relations, which can also have any of these storage structures. In this section we describe how to extend INGRES to include an R-tree storage structure that will be useful both as a primary structure for relations and for secondary indexes.

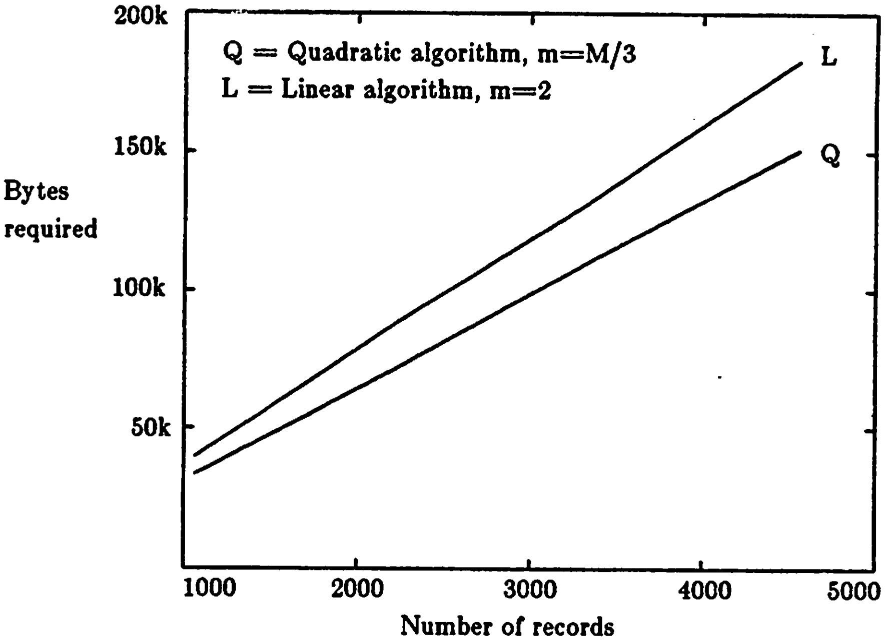


Figure 4.12
Space required for R-tree vs. amount of data.

4.5.1. Structure of R-tree Relations

Like an ISAM relation in INGRES, an R-tree relation has an index part and a leaf part, which are treated differently. The index part is visible only to the access methods, but the leaf part contains tuples, which are available to higher levels of the system and to the user. Pages for both parts are allocated and freed dynamically. Each index page contains:

(1) A collection of *entries*. Each entry contains a) a pointer to another page, a child node in the index tree, and b) a representation of an n-dimensional rectangle consisting of high and low values for each dimension.

(2) An entry map, telling which entry slots are occupied.

(3) An entry count, indicating how many entry slots are occupied. This is useful to have, although it is redundant.

Each leaf page contains:

(4) A collection of tuples.

(5) A tuple map, telling which tuple slots are occupied.

(6) A tuple count.

In the case of an R-tree relation used as a secondary index in the manner of the preceding sections, tuples stored in the leaf part closely resemble entries in nodes of the index part. Each leaf tuple contains a tuple identifier for a tuple in the primary relation, and a description of a rectangle enclosing the spatial object represented by the primary tuple. This description consists of $2n$ columns, for the high and low coordinate values of the rectangle along all dimensions. The difference between index entries higher in the R-tree and leaf tuples is that index entries contain pointers to other pages in the same relation, whereas leaf tuples contain identifiers of tuples in the primary relation.

When a primary relation is structured as an R-tree, tuples in the leaf part are ordinary data tuples. One column of the relation is designated to play the part of each rectangle boundary, and values from these columns are used to build the index part of the relation. Other columns can contain data unrelated to the index structure. Note that the number of tuples that will fit in a leaf page may be different from the the number of index entries in a higher page, but this does not affect how the tree is used for searching or interfere with maintenance of its structure.

4.5.2. Declaration Syntax

A normal QUEL statement of the form

index on relname **is** indexname (dom1, dom2, ... dom2n)

serves to set up a secondary index. Any relation, including one used as a secondary index, can be structured as an R-tree by a QUEL statement of the form

modify relname **to rtree on** (domain1, domain2, ... domain2n).

An even number of domains must be listed, and they are interpreted as representing rectangle boundary values in the order low-x, high-x, low-y, high-y, etc. Note that not all domains in the relation need to participate in the indexing structure, and the same domain can be used more than once. For example, the same domain could be used for both low-x and high-x, if the objects in question take exact values on the x-axis instead of spanning intervals there.

4.5.3. System Catalogs

INGRES has a number of special relations that store catalog information used by the system. One is the *relation* relation, which indicates, among other things, which of the available storage structures is used for each relation. No changes are required in the *relation* relation to support an R-tree storage structure except the obvious one of allowing one more storage structure. The *attribute* relation contains a tuple for each column of every relation. One column tells whether the attribute represented by each tuple is part of a key in its relation, and if it is part of a composite key, its ordinal place in the key. This mechanism can be used for R-tree relations to designate columns in tuples of the leaf part that are to be rectangle boundary values. The only addition necessary is a convention that key number 1 represents the low value in the first dimension, number 2 the high value, number 3 the low value in the second dimension and so on.

Current INGRES has a special relation called *indexes* in which is stored a tuple for each secondary index. It is organized by the name of the indexed primary relation, in order that useful indexes can be found quickly during processing of queries on the primary relation. A tuple in the *indexes* relation contains six slots for designating fields of the primary that make up the index key. The basic structure of this relation does not need to be changed for R-tree indexes, although the number of key fields allowed should be increased to perhaps twelve. Six slots allows a maximum of three dimensions for R-tree rectangles; this is useful but is too tight a maximum. Six dimensions allows more flexibility. The same convention as used in the *attribute* relation regarding order of the keys can be applied to the *indexes* relation.

4.5.4. Query Processing Code

An R-tree structure is useful for processing any query with a qualification restricting the range of one or more keys used as a rectangle boundary. Such a query can be processed in the normal way until the routine *strategy* is called. This routine examines the qualification and determines how the relation structure or any secondary indexes can be used. It also sets values for the keys, so that the function *find* will be able to determine the range of tuples to be examined at the start of a scan. *Strategy* must be augmented slightly for R-trees, so that it collects together all terms in the query qualification applicable to a particular R-tree structure, in effect building up the definition of a search rectangle. Sides of the search rectangle not specified by any qualification term should be considered to lie at plus or minus infinity. A partly defined search rectangle may still be worth using, because it can narrow the search to a part of the relation.

In current INGRES any scan of a relation is a two-step process. First, the routine *find* determines the range of the scan. *Find* uses the information assembled by *strategy* about relation structure and key values and returns the identifiers of the first and last tuples that should be examined. The two tuples can be in either a primary relation or an index. During the second step, the routine *get* is called each time a tuple is needed, and it simply iterates through the specified range of tuples. If the tuples are in an index, *get* retrieves an index tuple first, then the primary tuple it refers to.

This method does not work well for R-tree relations, because the tuples required are not necessarily in one continuous run. One solution would be to change *find* so that it returns a list of tuple ranges, or even a list of single tuples. *Get* can then process the list piece by piece.

A better approach would be to traverse the R-tree structure incrementally as new tuples are needed. A scan would still require two steps, but they would involve different logical functions than the ones performed by *find* and *get* in current INGRES. First, one routine would initializes the scan and allocates a descriptor for its state. Then, as each tuple was requested, a second routine would traverse the R-tree until another qualifying tuple was found, then save the state of the scan before returning. *Find* and *get* could be modified to perform these functions, as a way of incorporating R-trees into INGRES easily, but this would introduce an inconsistency into the logic of the program.

INGRES's *insert* and *delete* routines will need additional code to update R-tree indexes when tuples are added to or deleted from a relation. The necessary algorithms have been given above; incorporating them into INGRES should be straightforward.

CHAPTER 5

Abstract Indexes

5.1. Abstract Data Types

Computer aided design applications often involve special-purpose data objects, such as representations of rectangles and polygons, that are not supported as primitive types in database systems. The idea of abstract data types, originally developed in the context of programming languages, can be adapted to allow a database system to provide special-purpose types for specific applications. In this chapter we develop ways to process abstract data types efficiently in a database system.

An Abstract Data Type (ADT) is an encapsulation of data structures to represent objects of a particular kind, along with routines to implement all necessary operations on them. An abstract data type is usually built by an expert, who designs data structures to represent it and writes routines to manipulate the structures. Users can refer to objects of the type without concerning themselves with the details of how the objects are represented or manipulated, just as they use integers in arithmetic expressions without thinking about how they are stored or how addition and subtraction are performed. Abstract data types were developed in a programming language context in order to extend the usefulness of general-purpose programming languages into new application areas by allowing them to handle objects of new types that had not been anticipated in the original design of the language. A second goal was to allow programmers to think on higher levels of abstraction, concerning themselves only with objects of higher types and not with the details of their representation.*

Carrying this idea over to database systems, it is possible to augment a DBMS's built-in type system with a facility for defining new kinds of data objects to suit particular applications, and new operators to go with them. The database system can store and retrieve abstract data type objects, calling upon user-supplied routines to perform any required operations on them and otherwise processing queries with its usual mechanisms. Clearly this will make a general-purpose database system more useful in new application areas, especially where commonly-used data objects have complex internal structure. The second goal of abstract data types is also fulfilled, because database users can call for the storage and retrieval of high-level objects of the types they really use, without having to decompose them into parts of more primitive types. In VLSI CAD, for example, it would be useful to have objects of type *box* and *transform-matrix* (see Chapter 2).

Fogg and Ong have added an abstract data type facility in INGRES; their work is described in [28,74]. With their implementation a user can define new data types, use them in columns of relations, and specify new operators that are implemented by user-supplied routines. However their implementation does not allow the database system to construct indexes or other efficient access structures for relations based on values in abstract data type columns, or to optimize the processing of queries that

* For a discussion of abstract data types in programming languages see Liskov [57], also Gries and Gehani [30].

involve operations on abstract data types. We build on their work by outlining a way to construct fast access paths based on the special semantics of abstract data type objects, and to make use of special semantics to optimize query processing.

5.2. Abstract Indexes

Traditionally, secondary indexes provide a fast access path to required data items when a query specifies an exact match with a user specified value or a comparison operator applied with a specified value. For example, the queries

```
range of e is EMPLOYEE
retrieve (e.salary) where e.name = "Jones"
retrieve (e.name) where e.salary > 1000
```

can be answered efficiently by using indexes on the *name* and *salary* fields respectively, if they exist.

Since new operators can be defined for normal data types and for new ADT data types, one would want the following capabilities:

1) Indexes on ADT columns with existing operators.

For example, consider the situation where salaries of employees are stored as packed decimal numbers. Since this is not one of the built-in data types in many systems, an ADT is required. One would want to index salary so that the above query could be answered effectively. In this case extending an indexed sequential access method to support the new data type will be adequate.

2) Indexes on normal columns using new operators.

For example, consider the query:

```
range of e is EMPLOYEE
retrieve (e.name) where e.name !! 7
```

which requests the names of employees whose names are exactly 7 characters long. The new operator !! counts the number of non-blank characters in a name and compares the result to an integer operand. One would want a secondary index for the !! operator so that this query could be efficiently answered. Clearly, an index which provided a bin for each possible length would be appropriate.

3) Indexes on ADT columns with new operators.

Suppose a BOXES relation is defined by

```
BOXES (box-num=int, box-desc=box-adt)
```

where *box-adt* is an abstract data type that describes a rectangle in two dimensions. Consider the query to find all the boxes that intersect the unit square at the origin:

```
range of b is BOXES
retrieve (b.box-desc)
          where b.box-desc || "0,0,1,1"
```

We need an index that will allow retrieval of only the boxes that qualify, or will at least restrict the search to a small subset of the BOXES relation.

The objective of this section is to propose a scheme which supports all three capabilities above. Any DBMS provides a selection of access methods, and ideally it should be easy to add new ones [91]. Our goal is to allow any access method to operate in the more general context of ADTs. Hashing and indexed-sequential (ISAM) are the access methods currently supported by INGRES, and we focus the discussion on extending these. As a running example, we use the BOXES relation defined above.

An index can be created using the INGRES index command:

index on BOXES is B-INDEX (box-desc)

This will create a relation of the following form which will be used as a secondary index:

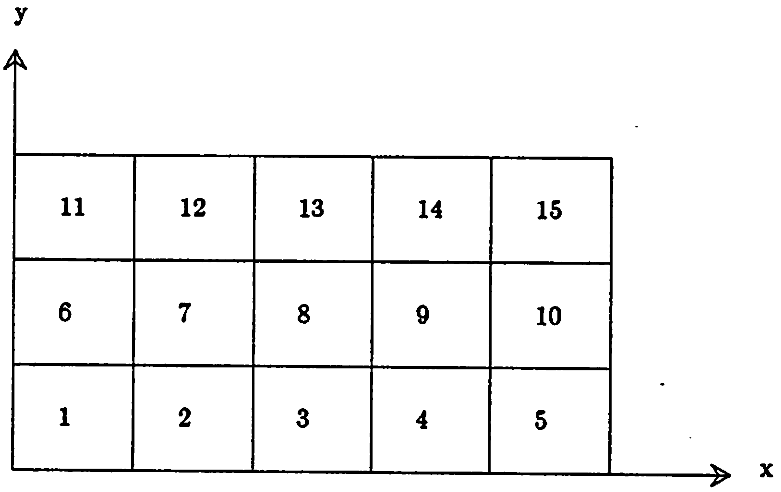
B-INDEX (box-desc, pointer-to-tuple)

A row exists in B-INDEX for each row in the BOXES relation and contains the box-desc field along with a pointer to the given tuple in BOXES. The index B-INDEX is initially stored as a heap and must be modified to hash or ISAM to be useful. For example:

modify B-INDEX to hash on box-desc using my-function

The only change to the current modify command is the inclusion of a "using" clause. INGRES normally builds hashed secondary indexes by allocating a number of buckets, then reading the tuples one by one, calling its internal hash function to obtain a bucket number and storing the tuple in the correct bucket. In this context INGRES calls my-function instead of its built-in hashing function to obtain bucket numbers. My-function must be a valid function registered through the ADT registration facility which expects a box-adt as an argument and returns an integer. No other modifications are required to the code if my-function returns a single integer.

However, suppose we have a grid in the x-y plane as shown in Figure 5.1, and we want my-function, when passed a box, to return the numbers of all the grid cells that it intersects. Grid cell zero is reserved for boxes which extend outside the boundary of



A grid structure for my-function
Figure 5.1

the above structure. In this situation my-function returns a list of buckets instead of a single bucket number and INGRES must insert a row in the appropriate bucket in B-INDEX for each value in the list. The modify command for this structure is

```
modify B-INDEX to hash on box-desc
          using my-function (param-list)
```

Here param-list is a character string containing necessary information such as the number and size of the grid squares and the location of the grid in the plane. These values could be hard-wired into my-function, but it is preferable that they be setable for each index.

We now illustrate how to use an ISAM structure with new columns and operators. Again, we could run the following modify command:

```
modify B-INDEX to isam (box-desc) using <+
```

Normally, an ISAM structure is built by sorting the values for box-desc using the built-in operator < to define the sort order. In this case the index can be built in an analogous way by substituting the operator <+ to define some ordering on boxes, for example by comparing their areas. <+ would be expected to compare two box descriptions and return true or false if one was "less than" the other. The ISAM structure would then support the ordering determined by <+.

Once a hashed or ISAM secondary index is created for the boxes relation, one must specify to INGRES how the index can be used in processing queries. INGRES has a built-in function, FIND, for hashed structures which will return the hash buckets which must be inspected for tuples which satisfy a particular query. In the current implementation a hash bucket is identical to a UNIX page, so FIND returns a collection of pages. An analogous FIND function returns a collection of pages for an ISAM structure. These functions are called by specifying the value used in a qualification and the operator involved. For example, for the qualification

```
where e.salary > 1000
```

FIND is called with parameters > and 1000. In our extended environment, a FIND function must be provided for each possible operator for which the index can be used. We propose a new INGRES command for the purpose, i.e.

```
use B-INDEX with find-function
        for (|| box-adt, box-adt ||)
```

This command specifies the circumstances under which the the routine, find-function, should be called to provide the required collection of pages to search. The above example indicates that this function is appropriate when the intersection operator || is encountered connecting a variable and a value of type box-adt. Moreover, the value can be on either side of the || operator. For example, suppose one submitted the query:

```
range of b is boxes
retrieve (b.all)
where b.box-desc || "0,0,1,1"
```

The string on the right is converted automatically to an object of type box-adt because || is defined to take box-adt arguments. After the conversion, the qualification is of the form

```
where b.box-desc || box-adt
```

and therefore B-INDEX can be used to process the query. The ADT function find-function is called to return a list of pages which must be examined. Then, INGRES simply iterates over the list examining each index entry, following the appropriate pointer, obtaining a tuple from boxes and finally evaluating the user's qualification to ascertain if it is satisfied for the tuple in question.

It is possible to define different FIND routines for different operators as illustrated below. Suppose one defines a new operator "#|" which compares a box and a line and returns true if the box is "to the left of" the line. The index B-INDEX can be used to process queries involving the #| operator; however, a new FIND function must be used:

```
use B-INDEX with second-fn for (#| line-ADT)
```

A user can submit a query such as

```
retrieve (b.all)
where b.box-desc #| "0,0,1,3"
```

whereby he wants to see all boxes which are to the left of the line from (0,0) to (1,3). If the grid structure for B-INDEX from Figure 5.1 is one unit long on each side, then the boxes which qualify must lie in grid cells 1, 6, 11 or 0 and the others can be excluded. The function second-fn can provide the needed semantics.

When more than one index can be used to process an INGRES query, e.g.

```
where b.box.desc || "0,0,1,1"
    and b.box.desc #| "0,0,1,3"
```

then INGRES must choose which index to use in processing the query. This is currently done by a hard-wired strategy routine. To be able to choose in the above context, this routine must be generalized to call both find functions to obtain list of pages and then compare the sizes of the results, choosing the smaller list for iteration.

5.3. R-trees with Abstract Indexes and New Operators

Implementation of an R-tree access method in INGRES as outlined in the preceding chapter should combine well with abstract indexes. For example, suppose the tuples of a relation represent geometric objects whose limits along the various dimensions are not given directly by the values of columns, but must be derived from the value of an abstract data type. The user can supply a routine that calculates a covering rectangle from the abstract data type stored in a tuple, and INGRES can invoke the routine for each new tuple in order to determine a rectangle for use in an R-tree index.

An R-tree index may also be useful for queries containing new operators. An example is an "overlap" operator, which could be used to retrieve all the geometric objects overlapping a constant figure. An overlap query can be expressed much more compactly using such an operator than by listing ranges along each dimension separately. Furthermore, the geometric object and even the constant figure can be of abstract data types. A user-supplied routine, linked in by the method outlined above, can translate an overlap qualification into a list of ordinary inequality terms, which INGRES can use with an R-tree index to process the query.

CHAPTER 6

Multi-Valued Attributes

6.1. Introduction

In database applications varying-sized collections of entities of the same type often need to be treated as composite objects. Examples are committees (which are collections of people), polygons represented as ordered collections of vertices, and integrated circuit cell designs as collections of layout rectangles. A collection of entities of the same type can be represented by an owner-coupled set in the DBTG data model by having records in the set correspond to elements of the collection, and letting the owner record stand for the composite object. However this data model has many disadvantages which have been well documented in the past [16]. Collections can also be represented in a normalized relational schema, in which each element becomes a tuple in a relation. Tuples belonging to a particular composite entity are bound together by having a common value in an "owner" field. For example, polygons can be represented by tuples in a POLYGON relation, and their component vertices by tuples in a VERTEX relation:

POLYGON (polygon-num, . . . other data . . .)
VERTEX (polygon-num, vertex-num, x-coord, y-coord)

Each tuple in the POLYGON relation stands for a polygon whose vertices are represented by multiple VERTEX tuples sharing the same value in the *polygon-num* column. The *vertex-num* column serves to order the vertices for each polygon.

This approach produces a "flat" database in which all objects, regardless of whether they are composites or components of some other object, are represented by tuples. A join is required to collect the elements of a particular composite. This is appropriate where the composite and its components both need to be independently accessible, as in the case of committees made up of professors. However in other cases component parts are rarely referenced independently from their composites, and the join to collect the group together makes queries unnecessarily complicated and slows processing. For example one rarely retrieves polygon vertices without first selecting the ones belonging to a particular polygon.

Objects whose components are rarely accessed separately can be modeled by relations whose attributes can take multiple values, so that component parts can be grouped directly in tuples representing the composites. This makes independent access to component objects more difficult but makes access to them as parts composites easier. If multi-valued relation attributes are available, then a relational query language like QUEL [34] can provide for searching within the database down to the level of tuples representing composite objects. Searching within multi-valued fields is also needed, and new features added for this purpose are likely to make the data access language excessively complicated unless searching within a collection can be done using almost the same language constructs as for searching in relations. In addition, because the number of components in a composite object may be very large, access methods of the kind used for relations are needed to make searching within them efficient. As an example, geographic databases often store large map objects like the outlines of countries as single polygons with thousands of vertices. Searching through collections like these by linear scanning will clearly be very inefficient.

In this chapter we outline three ways to provide for set or array attributes in relations: by defining an abstract data type (ADT) "array-of", by providing built-in array types as part of the basic type system of the DBMS, and by allowing relations to nest. Additions to QUEL are presented for each plan. We show that the third alternative offers significant advantages in keeping the data definition and data access languages simple, while allowing easy implementation and efficient data access using existing mechanisms.

The database language GEM [101] includes provision for set-valued attributes of a kind that is somewhat different from what is needed here, and also supports aggregate objects similar to the composites described above. The differences between GEM's features and our third proposal is discussed in Section 6.5. Non first normal form relations have been the subject of previous writings by other authors who took different approaches; see [21,40,64].

6.2. Arrays as Abstract Data Types

Multiple data values in a single column of a relation can be provided by means of an "array-of" abstract data type defined as outlined by Fogg [28] and Ong [74] and parameterized by the type of the array elements. Thus we can declare an array of integers, of characters, or of some other abstract data type, including *array-of*. The definition of the *array-of* type and routines to perform all legal operations on objects of the type are implemented outside the database system and provided as a package to database users, so that arrays can be used in any application. We have chosen to use ordered arrays as an example, but the ADT mechanism works equally well for unordered sets if the application does not require ordering.

The following small set of operations on objects of the *array-of* type is required:

(1) A *read* function, to extract the value at a particular position in an array. This function can be invoked by the use of a column name in standard array notation as used in the C programming language [51], e.g. column-name[index], on the right of an equals sign or in another place where retrieval of a value is implied. The keyword ALL used in place of an index number indicates that all values should be retrieved, one after the other. Fogg and Ong's implementation of abstract data types does not provide for such elaborate syntax, but a fully developed ADT facility could.

(2) A *write* function, again invoked by standard array notation: column-name[index] = new value.

(3) An *insert* function, which adds a new value in the middle of an array and moves all higher-numbered elements up: column-name[index,INSERT] = new value.

(4) An *append* function, which adds a new element after the last occupied cell in an array: column-name[APPEND] = new value.

(5) A *delete* function, which causes all higher-numbered elements to move down: column-name[index] = DELETE. Again, ALL can be used in place of the index to delete all elements of an array.

As an example consider a relation describing airline flights. Each flight can have an arbitrary number of stops, characterized by a city name and arrival and departure times.

```
create FLIGHT (
        num = int,
        city = array-of(c20),
        arr-time = array-of(int),
        dep-time = array-of(int))
```

The queries below retrieve the first three stops of flight 99, and all stops.[*]

```
retrieve (FLIGHT.city[1], FLIGHT.city[2], FLIGHT.city[3])
        where FLIGHT.num = 99

retrieve (FLIGHT.city[ALL]) where FLIGHT.num = 99
```

We can change the third stop from Las Vegas to Reno with

```
replace FLIGHT (city[3] = "Reno") where FLIGHT.num = 99
```

The following query adds a new second stop to flight 99, and causes the old second and later stops to move to higher-numbered positions in the array:

```
replace FLIGHT (
        city[2,INSERT] = "Boulder",
        arr-time[2,INSERT] = 1030,
        dep-time[2,INSERT] = 1045)
```

We can also provide aggregates for numeric arrays:

```
array-min(array name)
array-max . . .
array-count . . .
array-avg . . .
array-sum . . .
array-prod . . .
```

[*] We have used the name of a relation as a tuple variable, which is not allowed in standard QUEL. This obvious addition to the language has been included in both GEM and a commercial version of QUEL [82].

The following query uses aggregates to find the total duration of flight 99:

```
retrieve (duration =
          array-max(FLIGHT.arr-time) - array-min(FLIGHT.dep-time))
          where FLIGHT.num = 99
```

Access to array elements by index is not very useful for searching within an array, although it is possible to put a query inside a program loop in order to examine the elements one by one. An alternative is to provide searching functions along with aggregates as part of the *array-of* type, but this is equivalent to inventing a second query language just for use within arrays. Users of the database system would have to use the database query language to locate tuples containing arrays of interest, then switch to another language to select the desired elements. Probably the two languages would be mixed within a query.

The power of using ADTs with a database system lies in their ability to support special-purpose functions that are not normally provided by a general-purpose DBMS. As an example, consider a CAD application that uses polygons, and represents them by arrays of vertices stored in a relation:

POLYGON (poly-num = int, vertex = array-of(point-adt))

Here *point-adt* is another abstract data type, previously defined, containing the coordinates of a point in two dimensions. In this application it is quite possible that the areas of polygons are important, but calculating them from ordered arrays of points using a general-purpose query language would be difficult. A routine to perform the calculation would logically be implemented as part of the application, and then used to organize the POLYGON relation in an isam structure ordered by area, using methods described in Fogg [28] and Ong [74] and in the preceding chapter:

```
modify POLYGON to isam on vertex using area-function
```

Then operators could be defined for use within QUEL to help process queries referring to the areas of polygons. For example a "bigger-than" operator, denoted by >>, could be defined and used to find all polygons with area greater than 100:

```
retrieve (POLYGON.poly-num) where POLYGON.vertex >> 100
```

This example illustrates how special semantics can be used to search within relations efficiently in order to find tuples with qualifying arrays. We can provide similar semantic routines to help search within arrays. Continuing the above example, functions can be implemented to find the vertices of a polygon which are farthest to the right, to the left, up and down:

```
max-x(array-of(point-adt))
min-x . . .
max-y . . .
min-y . . .
```

These functions take arguments whose type is *array-of(point-adt)* and return results of type *point-adt*. It is easy to imagine situations in which special routines perform non-trivial calculations on large arrays stored in specially designed and highly tuned storage structures. The package would outperform any combination of general-purpose database schemas and queries for particular searching jobs, but each one would require its own functions and data structures.

According to the ADT paradigm, design of the internal storage structure of an array ADT is entirely up to the implementor of the type and is handled outside the database system. In the case of variable arrays of unbounded size, it would be impractical to try to store the array data itself in tuples. Instead, the tuple would contain a pointer, file name or some other identifier and the data would be stored externally in a file. It would not be possible to make use of storage structures provided by the DBMS for relations, like B-trees and hashing, or secondary indexes. All search mechanisms and supporting structures would have to be implemented from scratch. The DBMS's query processing and optimization capabilities also would not be available, nor could array data be protected by the database system's crash recovery mechanisms, because it would be stored outside the database.

Use of ADTs allows a good division of work between a DBMS and a set of application programs. The DBMS provides efficient data storage and retrieval along with other facilities, while application-dependent semantics are handled in the application. In addition, semantic routines developed as part of the application can be used to help the DBMS process queries. ADT-arrays are most useful for relatively small composite objects which have special semantics, where little searching within objects is required. For extremely complex objects, ADTs become less practical because the routines that must be supplied to unpack the component parts become numerous and difficult to use, and they duplicate the functions of a query language. For extremely large arrays sophisticated methods of internal searching become necessary and this requires duplication of the DBMS's query processing functions.

6.3. Built-In Array Types

A second way to allow a varying number of data values to occupy a column of a relation is to build array types directly into the database system. In this section we present syntactic extensions to QUEL for defining array columns in relations and for expressing queries that refer to individual array elements.

Array columns can be declared using square brackets with syntax similar to that of C. The relation mentioned earlier describing airline flights is defined as follows:

```
create FLIGHT(num=int, city[ ]=c20, arr-time[ ]=int, dep-time[ ]=int)
```

The empty brackets [ ] indicate array columns with no pre-determined number of elements. A number inside the brackets would indicate a fixed number of elements. Individual elements of arrays can be addressed using constant indexes; for example the following queries (identical to the ones for ADT arrays) print the names of the first three cities visited by flight 99, and change the third stop to Reno:

```
retrieve (FLIGHT.city[1], FLIGHT.city[2], FLIGHT.city[3])
        where FLIGHT.num = 99

replace FLIGHT (city[3] = "Reno") where FLIGHT.num = 99
```

The update operators that took the form of user routine calls for ADT arrays are now part of the DBMS. *Append, insert* and *delete* can be implemented with the same syntax and semantics as proposed for ADT arrays, and the query to add another stop to flight 99 looks the same:

```
replace FLIGHT (
        city[2,INSERT] = "Boulder",
        arr-time[2,INSERT] = 1030,
        dep-time[2,INSERT] = 1045)
```

The addition of another new feature, *index variables,* extends QUEL to allow it to search within arrays in a way that is very similar to the way it searches in relations, and makes possible operations that are not feasible with ADT arrays. An index variable is similar to a tuple variable but ranges over elements of an array instead of tuples in a relation. It is represented by a letter or name that is not a keyword and is not otherwise used in query, and is declared implicitly by being used to index an array. All uses of an index variable within a query are bound to each other. An index variable would be used to list all elements of an array, in ascending index order:

```
retrieve (FLIGHT.city[i], FLIGHT.arr-time[i], FLIGHT.dep-time[i])
        where FLIGHT.num = 99
```

In each line of the result, *i* has the same value for all arrays, so that all data for a particular stop is printed on the same line. Thus index variables serve to iterate over arrays and also to coordinate indexes used in several arrays simultaneously. Both these functions parallel the use of tuple variables with relations. Iteration over ADT arrays is achieved in a crude way by use of the ALL keyword, but coordination between arrays is not possible.

Index variables can appear in qualification clauses by themselves or as indexes for arrays, as in the following example which prints the numbers of all flights arriving in St. Louis after at most two earlier hops:

```
retrieve (FLIGHT.num)
          where i <= 3 and FLIGHT.city[i] = "St. Louis"
```

Note that extended QUEL with index variables provides both direct and associative access to array elements. Direct access is achieved by using specific numbers as array indexes, and selection of array elements by value is accomplished by the use of variable-indexed arrays in qualification clauses.

Numeric aggregates similar to those proposed for ADT arrays can be defined and made part of the query language, just as aggregates over relations are included in current QUEL.

```
array-min (numeric expression where qualification)
array-max . . .
array-sum . . .
array-avg . . .
array-prod . . .
```

Index variables make it possible to include qualification clauses and to refer to more than one array element in a single expression. Two additional boolean valued aggregates are useful, *all* and *any*.

```
all (boolean expression where qualification)
any (boolean expression)
```

All takes a boolean expression as argument, and returns true if all array elements satisfying the (optional) qualification clause caused the expression to be true. For example,

```
all(FLIGHT.num < 300 where FLIGHT.city[i] = "Dallas")
```

returns true if all flights through Dallas are numbered less than 300. *Any* returns true if there is some array element or combination of elements that makes the expression true. A separate qualification clause is unnecessary because the qualification can be included as part of the main boolean expression. For example,

```
any(FLIGHT.city[i] = "Dallas" and FLIGHT.num = 99)
```

returns true if flight 99 passes through Dallas, and

```
any(FLIGHT.city[i] = "St. Louis"
        and FLIGHT.city[j] = "Kansas City"
        and i < j)
```

returns true if there is a flight going from St. Louis to Kansas City. The last example also illustrates the use of more than one index variable in an aggregate. Two more examples examples further demonstrate the power of index variables combined with QUEL.

(1) Print the starting and ending times of the longest-running flights.

```
range of f1 is FLIGHT
range of f2 is FLIGHT
retrieve (f1.num, f1.dep-time[i], f1.arr-time[j])
        where f1.arr-time[j] - f1.dep-time[i] =
                        array-max(f2.arr-time[k] - f2.dep-time[m])
```

(2) Calculate the earliest time it is possible to reach Houston starting from Atlanta after 8 am., changing planes once:

```
range of f1 is FLIGHT
range of f2 is FLIGHT
retrieve (arrival-time =
        array-min(f2.arr-time[m]
                        where f1.city[i] = "Atlanta"
                        and f1.dep-time[i] > 0800
                        and i < j
                        and f1.city[j] = f2.city[k]
                        and f1.arr-time[j] < f2.dep-time[k]
                        and k < m
                        and f2.city[m] = "Houston"))
```

Arrays can be nested, as in the following relation representing integrated circuit wires as collections of conducting runs, each run described by the sequence of points it passes through. To make the example more readable, we first define a symbol POINT which stands for the declarations required for a single point. Then we use the symbol in a QUEL statement to create a WIRE relation:

```
macro POINT (x=float, y=float)
create WIRE (wire-num=int, run[ ]=(run-num=int, point[ ]=POINT))
```

The following query lists all points in wire 99, run by run. The nesting of loops for iterating over arrays is determined by the nesting of the arrays they index, or by the order of their appearance if there is an ambiguity.

```
range of w is WIRE
retrieve (w.run[i].run-num, w.run[i].x[j], w.run[i].y[j])
        where w.wire-num = 99
```

Multiple index variables that are not bound to each other can be used to generate a cross product. In the relation below each tuple compactly represents a set of grid crossing points for a map, by storing two vectors of coordinates for the longitude and latitude lines. The following query generates the coordinates of all grid crossing points as the cross product of the vectors:

```
create GRID (map-num=int, longitude[ ]=float, latitude[ ]=float)

retrieve (GRID.longitude[i], GRID.latitude[j])
        where GRID.map-num = 99
```

An array stored in a database is accessible only through its parent tuple, and this means that a query must be processed in two parts: First, the database system must plan and executes the main query, finding qualifying tuples corresponding to all tuple variables. Then for each set of tuples it must apply those parts of the query dealing with arrays. The second phase of this process may even use different strategies for different arrays, depending on their sizes and other characteristics.

It would be desirable to store arrays with their parent tuples, in order to insure fast access through them. In general this is impractical because arrays may be very large, and placing them among tuples of the relation would spread the tuples out over large areas of storage and make scans over the relation inefficient. Furthermore arrays can grow after their parent tuples are in place, and this complicates storage allocation. Therefore it is probably best to store arrays in a separate place, still under control of the database system, and keep pointers to them in the parent tuples.

If arrays can be presumed to be fairly small, or if it is rare to access only subsets of their elements, then the elements can be stored consecutively in index order. In this case the processing of subqueries within arrays would be based on full scans of all the elements. More sophisticated array storage structures would allow more efficient query processing, but language constructs to specify them would complicate the data definition language considerably. The same applies to secondary indexes on arrays.

Providing arrays as part of the standard type definition mechanism of a database system is justified by the fact that they are useful in a wide variety of applications. The query language for manipulating data inside arrays is very similar to standard QUEL, is comparably powerful and only moderately more complicated. Unfortunately the wide variety of physical storage structures available for relations cannot be used for arrays, so searching may often be less efficient than in relations. Moreover it is not possible to incorporate application-dependent semantics in a way that speeds processing, as is possible with ADT arrays. Built-in arrays can be protected by the database system's crash recovery facilities because they are stored inside the database, unlike ADT arrays.

6.4. Repeating Fields as Nested Relations

Let us consider representing multi-valued columns as relations instead cf arrays. This allows a simpler user interface than is possible with either ADT arrays or built-in array types, because most of the language constructs for schema definition and data access can be adapted from standard QUEL with very few changes.

An important difference between relations and arrays as described above is that arrays are ordered and relations are not. Order plays an important role in some composite object representations, e.g. polygons as sequences of vertices, but in other cases it is not important, for example for a VLSI circuit composed of layout rectangles. If order is required it can be supported by the addition of an extra column to a relation or by the use of an ordered relation [59,63].

Definition of nested relations is accomplished by using parentheses to nest the lists of column definitions in QUEL *create* statements. Our earlier example involving airline flights becomes

```
create FLIGHT (num=int,
        stop=(num=int, city=c20, arr-time=int, dep-time=int))
```

Dot notation is used to reference specific columns of nested relations:

```
retrieve (FLIGHT.stop.city)
        where FLIGHT.stop.num <= 3
        and FLIGHT.num = 99
```

The following alternative form for the same query illustrates how range variables are used with nested relations. When a tuple variable *f* takes on the value of a tuple in the FLIGHT relation, then *f.stop* is a relation, and therefore another tuple variable can range over it:

```
range of f is FLIGHT
range of s is f.stop
retrieve (s.city) where s.num <= 3 and f.num = 99
```

Variable *s* ranges over a different relation for each value of *f*, and is always bound by the current value of *f* during processing of a query.

Changing the third stop of flight 99 from Las Vegas to Reno is done by applying a *replace* command to a tuple in a nested relation:

```
replace FLIGHT.stop (city="Reno")
        where FLIGHT.num = 99 and FLIGHT.stop.num = 3
```

To add a new second stop, it is necessary to renumber all the later stops. This would be done automatically with ordered relations as defined by Lynn [63].

```
replace FLIGHT.stop (num=num+1)
        where FLIGHT.num = 99 and FLIGHT.stop.num >= 2

append to FLIGHT.stop (num=2, city="Boulder", arr-time=1030, dep-time=1045)
        where FLIGHT.num = 99
```

Multiple tuple variables can be used just as in standard QUEL, as in this query to find all the flights going from St. Louis to Kansas City:

```
range of f is FLIGHT
range of s1 is f.stop
range of s2 is f.stop
retrieve (f.num)
        where s1.city = "St. Louis"
        and s2.city = "Kansas City"
        and s1.num < s2.num
```

Variables *s1* and *s2* are constrained to range over the same nested relation because they are both derived from *f*. Independent nested relation variables would be used to find two flights passing through Yellow Knife:

```
range of f1 is FLIGHT
range of s1 is f1.stop
range of f2 is FLIGHT
range of s2 is f2.stop
retrieve (f1.num, f2.num)
        where s1.city = "Yellow Knife"
        and s2.city = "Yellow Knife"
```

Or, equivalently:

```
range of f1 is FLIGHT
range of f2 is FLIGHT
retrieve (f1.num, f2.num)
        where f1.stop.city = "Yellow Knife"
        and f2.stop.city = "Yellow Knife"
```

It is not necessary to redefine the aggregates already included in QUEL. The following query finds the first plane out of Boulder after nine o'clock, using the standard QUEL *min:*

```
range of f is FLIGHT
retrieve (FLIGHT.num)
        where FLIGHT.stop.city = "Boulder"
        and FLIGHT.stop.dep-time =
                        min(f.stop.dep-time
                        where f.stop.dep-time > 0900
                        and f.stop.city = "Boulder")
```

To illustrate deeper nesting we recast our earlier definition of the WIRE relation and the query to retrieve all runs in wire 99, as follows:

```
macro POINTREL (point-num=int, x=float, y=float)
create WIRE (wire-num=int, run=(run-num=int, point=POINTREL))

range of w is WIRE
range of r is w.run
range of p is r.point
retrieve (r.run-num, p.point-num, p.x, p.y)
        where w.wire-num = 99
```

Unfortunately this produces the points of each run in random order. To get them in sequence we would have to retrieve the points for each run into a temporary relation and sort by *point-num.*

The extensions to QUEL outlined above provide for the definition of database schemas incorporating descriptions of set-like composite objects, and for easy access to them. The power for searching among components of a composite object is the same as for top-level relations. Additions to the language are minor and should be easy for users to learn. There are no pairs of parallel language constructs for searching in relations and in composites, as is necessary for both ADT and built-in arrays. Language for defining indexes and storage structures can also be applied to nested relations in the obvious way.

Nested relations can be stored in the same way as top-level relations, in order to take advantage of existing lower-level database facilities. Thus instances of nested relations are stored as separate relations, which are accessible only through their parent tuples. Catalog information that applies to all instances can be stored in system catalogs as usual and fetched once when a query is about to be processed. Information that varies from one instance to another, such as the size of a nested relation instance and the disk address where it is stored, can be kept in its parent tuple in order to avoid repeated references to system catalogs during processing of a query.

Storage structures and access methods for relations are very well developed, and can be used immediately with nested relations, whereas arrays require different techniques that are foreign to relational database systems. A similar body of knowledge exists about how to automatically optimize the processing of relational queries by making use of existing access paths, characteristics of stored data, and query structure. This also can be applied to nested relations, but arrays require new methods. For this reason we expect that nested relations will scale well to very large objects and perform better than arrays. In addition a nested relation system should be relatively easy to implement because so much of the necessary machinery is already in place.

Special semantics for composite objects cannot be supported as well by either nested relations or built-in arrays as by abstract data types. Hence we expect that ADTs will be more effective for small, array-like composite objects that have important special meanings in particular applications.

6.5. GEM and Nested Relations

Set-valued attributes are allowed in GEM [101], but set elements must be of atomic types, and there is no way to access individual elements. The intent seems to be to support small sets whose elements do not need to be manipulated in the same way as other database entities. Only set operations like set equivalence and set inclusion, and aggregates like *count* are provided. Examples given include sets of colors in which products can be ordered. One might be interested in seeing an entire set (a dozen or so colors) or in determining whether it is included in another set. The situation is quite different for an integrated circuit containing hundreds of layout rectangles, among which we want to search, changing one rectangle at a time.

Smith and Smith define an aggregate as a relationship between entities that is promoted to be an entity itself [89]. An example is a hotel reservation, which is a relationship between a person, a hotel, and a date. This kind of aggregate differs from the composites defined in this chapter, which are variable-sized collections of objects of the same type. GEM supports aggregates as defined by Smith and Smith, and uses dot notation to refer to participating lower entities. For example, given the schema

```
PERSON (p-name=c20, p-address=c50)
HOTEL (h-name=c20, h-address=c50, rate=int)
RESERVATION (guest=PERSON, accommodation=HOTEL, date=c12)
```

we can find the address of the hotel where Jones is staying on July 4th with the query

```
retrieve (RESERVATION.accommodation.h-address)
        where RESERVATION.guest.p-name = "Jones"
        and RESERVATION.date = "July 4"
```

This use of dot notation is very similar to the way we have used it to refer to component elements of collections. The two meanings do not conflict; in fact they can coexist quite nicely as illustrated by the example below. The schema can store descriptions of parts, manufacturers and supplies. A supply is a relationship between a part and a manufacturer and also includes a nested relation containing the available sizes of the part and the corresponding prices.

```
PART (p-num=int, descr=c20)
MFR (m-name=c20, address=c40)
SUPPLY (source=MFR, item=PART, variety=(size=c10, price=float))
```

The following query retrieves the names of all companies that supply 2-inch bolts. GEM's dot notation is used to refer to the PART and MFR entities participating in a SUPPLY, and our dot notation addresses tuples in the nested *variety* relation.

```
retrieve (SUPPLY.source.m-name)
        where SUPPLY.item.descr="bolt"
        and SUPPLY.variety.size="2 in."
```

CHAPTER 7

Conclusions

7.1. Summary

The objective of this thesis has been to develop ways to make relational database systems more efficient and easier to use in support of computer aided design applications. Our approach has been to design a series of new features that could reasonably be added to a relational system that would make it more useful for CAD, and possibly for other non-business database applications. In this chapter we summarize the work that was done and suggest directions for future research.

It is generally recognized that data management is an important part of computer aided design. Database systems claim to offer data management services, but CAD system designers have found it difficult to use them effectively because they are cumbersome, slow, and do not offer the services needed [88]. This thesis began with a discussion of how data management has been handled in a number of CAD systems, including some that used commercial database systems. Then in Chapter 2 we described an experiment in which we used INGRES to store VLSI CAD data and tested its performance on typical CAD retrieval operations. The goal of the experiment was to find performance bottlenecks and to look for ways in which INGRES was difficult to use, so that we could develop improvements.

We found that retrieval of hierarchically structured VLSI designs was inefficient and hard to program because it required transitive closure operations, which current INGRES does not support. We have developed a way to provide for transitive closure queries by adding an *append** command, and this was the subject of Chapter 3.

INGRES performed poorly on retrieval of spatial data falling within a small area, for two reasons. First, it does not have access methods suitable for efficient retrieval of multi-dimensional overlapping data, and so spatial queries must be processed by sequential searches of relations. Chapter 4 presents a new index structure for spatial data called an R-tree, which has been shown to be effective for spatial searching and is suitable for implementation as a database access method.

The second reason for INGRES's poor performance on spatial searches is that spatial proximity is not supported semantically by INGRES, whose query language and processing facilities are oriented toward numbers and strings. As a result proximity queries are difficult to write and they run inefficiently. Spatial proximity can be viewed as one example of special semantics characteristic of a particular application, and in Chapter 5 we outline a way for a database system to use semantic routines that are part of any application to help it construct indexes and optimize query processing. New operators can also be defined to allow compact expression of queries involving special semantics. This work builds on that of Fogg [28] and Ong [74] who added abstract data types (ADT's) to INGRES.

Entities modeled in a computer aided design database often are composites made up of many small pieces of the same type, for example circuit designs consisting of layout rectangles. It is desirable to find a convenient way to manipulate such composites efficiently and still be able to access the component parts when necessary. In Chapter 6 we have discussed three ways to represent such composites: by using an ADT facility to create an array type, by building array types directly into the database system, and by allowing relations to nest. We have outlined language extensions and implementation requirements for each approach, and have argued that nested relations provide the best combination of simplicity and power in the query language, with potential for compact and efficient implementation.

7.2. Directions for Future Work

Relational systems should be especially well suited to supporting the managerial and design-control functions of a CAD system. Examples are control of versions and revisions, and the partitioning of a design into segments to be delegated to different groups or people. Some work has been done in this area [46,47,61] and more is needed.

Our investigation of transitive closure operations was narrowly focused on the problem of expanding hierarchically structured designs. There is much more to be done with transitive closure, both in exploring how to apply it in a database context and in further development of implementation methods and query processing techniques.

The R-tree index structure would benefit from further development to make it as efficient as possible for actual use in a database system. It has been pointed out that better ways than linear scanning surely exist for searching within a node [66,98] and this should be investigated. In addition it would be interesting to look for less expensive methods for eliminating nearly-empty nodes than the one proposed.

Abstract data types and abstract indexes in a relational database seem to be very promising, but it is hard to assess their effectiveness without actually using them in an application. This would involve implementing ADT's and abstract indexes in a high-performance (i.e. non-prototype) relational DBMS and developing of a set of types and semantic routines for an application. A CAD system would be a good place to try this, but to our knowledge no such project is underway.

References

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

2. E. P. Allman, M. Stonebraker and G. D. Held, Embedding a relational data sublanguage in a general purpose programming language, *Proc. of a Conf. on Data: Abstraction, Definition, and Structure,* March 1976, 25-35. Special issue.

3. M. Arnold and J. Ousterhout, Lyra: A new approach to geometric layout rule checking, *Proc. 19th Design Automation Conference*, Las Vegas, Nevada , June 1982, 530-536.

4. M. Astrahan, et al., System R: Relational approach to database management, *ACM Transactions on Database Systems 1,* 2 (June 1976), 97-137.

5. R. Bayer and E. McCreight, Organization and maintenance of large ordered indices, *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, Houston, Texas, Nov. 1970, 107-141.

6. A. Beetem, J. Milton and G. Wiederhold, Performance of database management systems in VLSI design, *IEEE Database Engineering 5,* 2 (June 1982), 15-20.

7. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples,* McGraw-Hill, New York, 1971.

8. J. L. Bentley, Multidimensional binary search trees used for associative searching, *Communications of the ACM 18*, 9 (September 1975), 509-517.

9. J. L. Bentley, D. F. Stanat and E. H. Williams, Jr., The complexity of fixed-radius near neighbor searching, *Inf. Proc. Lett. 6*, 6 (December 1977), 209-212.

10. J. L. Bentley and J. H. Friedman, Data structures for range searching, *Computing Surveys 11*, 4 (December 1979), 397-409.

11. T. Beretvas, A general purpose multi-indexed data management system with history capabilities, Technical Report TR00.2078, IBM Data Systems Division Laboratory, Poughkeepsie, NY, 1970.

12. G. C. Billingsley, Program reference for KIC, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, August 1983. Also available as Electronics Research Laboratory Memorandum No. UCB/ERL M83/62.

13. M. Blasgen, et al., System R: An architectural update, Research Report RJ2581, IBM Research Laboratory, San Jose, California, July 1979.

14. D. Chamberlin, et al., Views, authorization, and locking in a relational database system, *Proc. AFIPS 1977 NCC 44*, (1977), 425-430.

15. P. L. Ciampi, et al., Control and integration of a CAD database, *Proc. 13th Design Automation Conference*, June 1976.

16. E. F. Codd and C. J. Date, Interactive support for non-programmers: The relational and network approaches, *ACM-SIGMOD Workshop on Data Description Access and Control*, May 1974, 11-41.

17. E. F. Codd, Extending the relational Model to capture more meaning, *ACM Trans. on Database Sys. 4*, 4 (1979), 397-434.

18. D. Comer, The ubiquitous B-tree, *Computing Surveys 11*, 2 (1979), 121-138.

19. D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, Implementation techniques for main memory database systems, *Proc. ACM/SIGMOD International Conference on Management of Data*, Boston, June 1984.

20. C. M. Eastman, System facilities for CAD databases, *Proc. 17th Design Automation Conference*, June 1980, 50-56.

21. A. Ege, *Non first normal form relations and engineering data bases*, Texas Tech University, Lubbock, Texas, c. 1982.

22. R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, Extendible hashing - A fast access method for dynamic files, *ACM Transactions on Database Systems 4*, 3 (September 1979), 315-344.

23. D. Fairbairn and J. Rowson, ICARUS: An interactive integrated circuit layout program, *Proc. 15th Design Automation Conference*, July 1978, 188-192.

24. S. J. Feldman, *Make — a program for maintaining computer programs*, Bell Laboratories, Murray Hill, 1978.

25. R. A. Finkel and J. L. Bentley, Quad trees - A data structure for retrieval on composite keys, *Acta Informatica 4*, (1974), 1-9.

26. D. Fitzpatrick, Cifplot — CIF interpreter and plotter, in *1983 VLSI Tools - Selected Works by the Original Artists*, R. N. Mayo, J. K. Ousterhout and W.

S. Scott, (eds.), Computer Science Division, EECS Dept., University of California, Berkeley, California, March 1983.

27. D. Fitzpatrick, R. N. Mayo, J. K. Ousterhout and W. S. Scott, eds., MEXTRA, Computer Science Report UCB/Computer Science Dept. 83/115, University of California, Berkeley, March 1983.

28. D. Fogg, Implementation of domain abstraction in the relational database system, INGRES, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, Sept. 1982.

29. H. Garcia-Molina, R. J. Lipton and J. Valdes, A massive memory machine, *IEEE Trans. on Computers c-33*, 5 (May 1984), 391-399.

30. D. Gries and N. Gehani, Some ideas on data types in high-level languages, *Comm. ACM 20*, 6 (June 1977), 414-420.

31. J. A. Guinea, *A High-Frequency Differential Narrow-Band Switched Capacitor Filtering Technique*, Ph.D. Thesis, University of California, Berkeley, June 1982.

32. A. Guttman and M. Stonebraker, Using a relational database management system for computer aided design data, *IEEE Database Engineering 5*, 2 (June 1982).

33. M. A. Harrison, *Introduction to formal language theory*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1978.

34. G. Held, M. Stonebraker and E. Wong, INGRES - A relational data base system, *Proc. AFIPS 1975 NCC 44*, (1975), 409-416.

35. K. Hinrichs and J. Nievergelt, The grid file: A data structure designed to support proximity queries on spatial objects, Nr. 54, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich, July 1983.

36. M. Hofmann, A method for topological compaction of programmed logic arrays, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, California, December 1980.

37. M. Hofmann and U. Lauther, HEX: An instruction-driven approach to feature extraction, *Proc. 20th Design Automation Conference*, 1983, 331-336.

38. M. Y. Hsueh, Symbolic layout and compaction of integrated circuits, ERL Memo UCB/ERL M79/80, University of California, Berkeley, December 1979.

39. B. Infante, D. Bracken, B. McCalla, S. Yamakoshi and E. Cohen, An interactive graphics system for the design of integrated circuits, *Proc. 15th Design Automation Conference*, July 1978, 182-187.

40. G. Jaeschke and H. J. Schek, Remarks on the algebra of non first normal form relations, *Proc. ACM SIGACT News-SIGMOD Conf. on Principles of Database Systems*, March 1982.

41. S. C. Johnson, YACC -- Yet Another Compiler Compiler, Computer Science Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

42. W. N. Joy, R. S. Fabry and K. Sklower, *UNIX Programmer's Manual, Seventh Edition, Virtual VAX-11 Version*, Computer Science Division, Dept. of EECS, University of California, Berkeley, June 1981.

43. J. Kaplan, Buffer management in a database environment, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, December 1979.

44. M. G. H. Katevenis, R. W. Sherburne, D. A. Patterson and C. H. Sèquin, The RISC II micro-architecture, *Proc. VLSI 83 Conference*, Trondheim, Norway, August 1983.

45. R. H. Katz, Performance enhancement for relational systems through query compilation, *Proc. AFIPS National Computer Conference*, 1979, 741-747.

46. R. H. Katz, A database approach for managing VLSI design data, *Proc. 19th Design Automation Conference*, June 1982, 274-282.

47. R. H. Katz and T. Lehman, Storage structures for versions and alternatives, Computer Sciences Technical Report #479, University of Wisconsin, Madison, July 1982.

48. K. H. Keller, KIC: A graphics editor for integrated circuits, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, June 1981.

49. K. H. Keller, A. R. Newton and S. Ellis, A symbolic design system for integrated circuits, *Proc. 19th Design Automation Conference*, June 1982, 460-466.

50. K. H. Keller, *An Electronic Circuit CAD Framework*, Ph.D. Thesis, Dept. of Elec. Engr. and Comp. Sci., University of California, Berkeley, 1984.

51. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

52. J. E. Kleckner, *Advanced Mixed-Mode Simulation Techniques*, Ph.D. Thesis, University of California, Berkeley, California, 1984.

53. W. Koenig and R. A. Lorie, *Storage of VLSI physical designs in a relational database*, IBM Research Laboratory, San Jose, California, 1982.

54. W. Koenig and R. Etienne, *A model for representing parameterized objects using a graphical editor*, IBM Research Laboratory, San Jose, California, 1982.

55. A. J. Korenjak and A. H. Teger, An integrated CAD data base system, *Proc. 12th Design Automation Conference*, June 1975, 399-406.

56. H. Landman, Automatic Layout of Optimized PLA Structures, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, June 1982.

57. B. H. Liskov and S. N. Zilles., Programming with abstract data types, *SIGPLAN Notices Notices 9*, 4 (1974).

58. G. M. Lohman, J. C. Stoltzfus, A. N. Benson, M. D. Martin and A. F. Cardenas, Remotely-sensed geophysical databases: Experience and Implications for Generalized DBMS, Computer Science Research Report RJ 3794 (43491) 2/17/83, IBM Research Laboratory, San Jose, California, February 17, 1983.

59. R. Lorie, R. Casajuana and J. Becerril, GSYSR: A relational database interface for graphics, IBM Research Report RJ2511 (32941), IBM Research Labs, San Jose, California, April 1979.

60. R. A. Lorie, Issues in database for design applications, IBM Research Report RJ3176 (38928) 7/10/81, IBM Research Laboratory, San Jose, California, July

1981.

61. R. Lorie and W. Plouffe, Complex objects and their use in design transactions, Computer Science Research Report RJ 3706 (42922), IBM Research Laboratory, San Jose, California, Dec. 8, 1982.

62. R. Lorie and W. Plouffe, Relational databases for engineering data, Computer Science Research Report RJ 3847 (43914), IBM Research Laboratory, San Jose, California, April 6, 1983.

63. N. Lynn, Implementing ordered relations in the relational database system INGRES, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, August 1982.

64. A. Makinouchi, A consideration on normal form of not-necessarily-normalized relation in the relational data model, *Proc. IEEE-ACM Conf. on Very Large Data Bases*, Tokyo, October 1977, 447-453.

65. R. N. Mayo, J. K. Ousterhout and W. S. Scott, eds., 1983 VLSI Tools - Selected Works by the Original Artists, Report No. UCB/CSD83/115, Computer Science Division, EECS Dept., University of California, Berkeley, California, March 1983.

66. R. McCord, private communication, April 1984.

67. T. McWilliams and L. Widdoes, Jr., SCALD: Structured Computer-Aided Logic Design, Technical Report No. 152, Digital Systems Laboratory, Stanford University, Stanford, California, March 1978.

68. T. McWilliams and L. Widdoes, Jr., The SCALD physical design subsystem, Technical Report No. 153, Digital Systems Laboratory, Stanford University, Stanford, California, March 1978.

69. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

70. L. W. Nagel, SPICE2: A computer program to simulate semiconductor circuits, ERL Memo ERL-M520, University of California, Berkeley, May 1975.

71. W. Newman and R. Sproull, *Principles of interactive computer graphics*, McGraw-Hill, New York, 1979.

72. A. Newton, D. Pederson, A. Sangiovanni-Vincentelli and C. Sequin, Design aids for VLSI: The Berkeley perspective, *IEEE Transactions on Circuits and Systems CAS-28*, 7 (July 1981).

73. L. A. O'Neill, et al., Designers workbench -- efficient and economical design aids, *Proc. 16th Design Automation Conference*, June 1979, 185-199.

74. J. Ong, The design and implementation of abstract data types in the relational database system, INGRES, Master of Science Report, Dept. Elec. Eng. and Comp. Sci., University of California, Berkeley, Sept. 1982.

75. J. Ousterhout, Caesar: An interactive editor for VLSI circuits, *VLSI Design II*, 4 (November, 1981), 34-38.

76. J. K. Ousterhout and D. M. Ungar, Measurements of a VLSI design, *Proc. 19th Design Automation Conference*, June 1982, 903-908.

77. J. K. Ousterhout, Corner stitching: A data structuring technique for VLSI layout tools, Computer Science Report Computer Science Dept. 82/114, University of California, Berkeley, 1982.

78. D. A. Patterson and C. H. Séquin, RISC I: A reduced instruction set VLSI computer, *Proc. Eighth International Symposium on Computer Architecture*, Minneapolis, Minnesota, May 1981, 443-457.

79. G. M. Pavlovic, Using a relational data base system to store text, Memorandum No. UCB/ERL M83/44, Electronics Research Laboratory, University of California, Berkeley, CA, July 1983.

80. M. L. Powell and M. A. Linton, Database support for programming environments, *Proc. Database Week Special Session on Engineering Design*, San Jose, California, May 1983, 63-70.

81. *ORACLE SQL Language - Reference Guide*, Relational Software Incorporated, Menlo Park, California, October 1980.

82. *INGRES Reference Manual, VAX/UNIX Version 2.0*, Relational Technology, Inc., Berkeley, California, June 1983.

83. K. A. Roberts, T. E. Baker and D. H. Jerome, A vertically organized computer-aided design data base, *Proc. 18th Design Automation Conference*, June 1981, 595-602.

84. J. T. Robinson, The K-D-B tree: A search structure for large multidimensional dynamic indexes, *ACM-SIGMOD Conference Proc.*, April 1981, 10-18.

85. M. Rochkind, The Source Code Control System, *IEEE Trans. on Soft. Eng. SE-1,* 4 (December 1975).

86. G. M. Sacco and M. Schkolnick, A mechanism for managing the buffer pool in a relational database system using the hot set model, Computer Science Research Report RJ-3354, IBM Research Laboratory, San Jose, California, January 1982.

87. J. L. Sanborn, Evolution of the engineering design system data base, *Proc. 19th Design Automation Conference,* June 1982, 214-218.

88. T. W. Sidle, Weaknesses of commercial data base management systems in engineering applications, *Proc. 17th Design Automation Conference,* June 1980, 57-61.

89. J. Smith and D. Smith, Database abstractions: Aggregation, *Comm. ACM 20,* 6 (June 1977), 405-413.

90. M. Stonebraker, Implementation of integrity constraints and views by query modification, *Proc. 1975 SIGMOD Conf.*, San Jose, California, June 1975, 65-78.

91. M. Stonebraker, E. Wong, P. Kreps and G. Held, The design and implementation of INGRES, *ACM Trans. on Database Sys. 1,* 3 (September 1976), 189-222.

92. M. Stonebraker, Hypothetical databases as views, *Proc. ACM SIGMOD Conference,* May 1981.

93. M. Stonebraker, J. Woodfill, J. Ranstrom, M. Murphy, M. Meyer and E. Allman, Performance enhancements to a relational data base system, Memorandum No.

UCB/ERL M81/62, Electronics Research Laboratory, University of California, Berkeley, CA, September 1982.

94. M. Stonebraker, H. Stettner, J. Kalash, A. Guttman and N. Lynn, Document processing in a relational database system, *ACM Transactions on Office Information Systems*, April 1983.

95. D. Weller and R. Williams, Graphics and relational data base support for problem solving, *Computer Graphics 10,* 2 (Summer 1976), 183-189, SIGGRAPH-ACM.

96. D. Weller and F. Palermo, Database requirements for graphics, *Proc. 18th IEEE Compcon*, February 1979, 231-234.

97. K. C. Wong and M. Edelberg, Interval hierarchies and their application to predicate files, *ACM Transactions on Database Systems 2,* 3 (September 1977), 223-232.

98. H. Wong, private communication, May 1984.

99. J. Woodfill, P. Siegel, J. Ranstrom, M. Meyer and E. Allman, *Ingres version 7 reference manual*, Dept. Elec. Eng. and Comp. Sci., U. C. Berkeley, 1981.

100. G. Yuval, Finding near neighbors in k-dimensional space, *Inf. Proc. Lett. 3,* 4 (March 1975), 113-114.

101. C. Zaniolo, The database language GEM, *SIGMOD Record 13,* 4 (May 1983), 207-218.

102. G. Zintl, A CODASYL CAD data base system, *Proc. 18th Design Automation Conference*, June 1981, 589-594.

103. M. Zloof, Query by example, *Proc. National Computer Conference 44*, (1975), 431-438, AFIPS Press.

104. M. Zloof, Query-By-Example: Operations on the transitive closure, IBM Research Report RC5526 (revised) (#24020), IBM Research Labs, San Jose, California, October 1976.

APPENDIX A

Test Programs for Chapter 2

Test 1: Top Level Geometry Retrieval

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include "macros.h"

#define CPUOHEAD 0.6
#define ELOHEAD 1

main(argc, argv)
int argc;
char *argv[];
{
##          char *cname, *dbname, *defstatus;
##          long cellid;

            dbname = *++argv;
            --argc;
            printf("\nProgram %s", "@(#)unexp.q                 1.10 2/11/83");
            printf("    database %s\n", dbname);

            while (--argc > 0)
            {
                    cname = *++argv;
                    cellid = NULL;
                    defstatus = "n";
##                  ingres dbname
##                  range of c is cell_master
##                  retrieve (cellid = c.master_id, defstatus = c.defined)
##                  where c.name = cname
##                  {
                              ;
##                  }
##                  exit
```

```
                        if (cellid == NULL || *defstatus != 'y')
                        {
                                printf("\n%s not found or undefined.\n", cname);
                        }
                        else
                        {
                                printf("\ncell %s, id is %ld\n", cname, cellid);
                                unexp(dbname, cname, cellid);
                        }
                }
                printf("\n");
        }

unexp(dbname, cname, cellid)
char *dbname, *cname;
long cellid;
{
##              long ecellid;
##              char *elname[6];
##              float ex1, ex2, ey1, ey2;
##              int i;
                extern int IItupcnt;
                int queries;
                long tuples;
                struct tms start, stop;
                float equeltime, ingrestime, cputime;
                long time(), estart, estop, etime;

                ecellid = cellid;
                elname[0] = "NM";
                elname[1] = "NP";
                elname[2] = "ND";
                elname[3] = "NC";
                elname[4] = "NI";
                elname[5] = "NG";

                /* start the clock */

                times(&start);
                estart = time(0);
                tuples = 0;
                queries = 0;
##              ingres dbname
##              range of r is cell_ref
##              range of b is box
##      range of w is wire
##      range of p is polygon

        /* collect geometries of each type in a temporary relation, */
        /* then retrieve one layer at a time for grouping. */

        printf("collecting boxes..\n");
##      retrieve into btemp (layer = b.use,
##              x1 = b.x1, y1 = b.y1, x2 = b.x2, y2 = b.y2)
##      where b.owner = ecellid
        queries++;

##      range of bt is btemp
        for (i = 0; i < 6; i++)
        {
##              retrieve (ex1 = bt.x1, ey1 = bt.y1, ex2 = bt.x2, ey2 = bt.y2)
##              where bt.layer = elname[i]
##              {
                        ;
##              }
                queries++;
                tuples += IItupcnt;
                printf("                %s layer, %d tuples\n", elname[i], IItupcnt);
        }
##      destroy btemp
        queries++;

        printf("collecting wires..\n");
##      retrieve into wtemp (layer = w.use,
##              x1 = w.x1, y1 = w.y1, x2 = w.x2, y2 = w.y2)
##      where w.owner = ecellid
        queries++;

##      range of wt is wtemp
        for (i = 0; i < 6; i++)
        {
##              retrieve (ex1 = wt.x1, ey1 = wt.y1, ex2 = wt.x2, ey2 = wt.y2)
##              where wt.layer = elname[i]
##              {
                        ;
##              }
                queries++;
                tuples += IItupcnt;
                printf("                %s layer, %d tuples\n", elname[i], IItupcnt);
        }
```

```
##          destroy wtemp
            queries++;

            printf("collecting polygons..\n");
##          retrieve into ptemp (layer = p.use, id = p.polygon_id,
##                    vnum = p.vertnum, x = p.x, y = p.y)
##          where p.owner = ecellid
            queries++;
##          modify ptemp to heapsort on layer, id, vnum
##          range of pt is ptemp
##          retrieve (ex1 = pt.x, ey1 = pt.x)
##          {
                      ;
##          }
            queries += 2;
            tuples += IItupcnt;
            printf("              %d tuples sorted & retrieved.\n", IItupcnt);
##          destroy ptemp
            queries++;

            /* stop the clock and print statistics */

##          exit
            times(&stop);
            estop = time(0);
            equeltime = (stop.tms_utime - start.tms_utime
                      + stop.tms_stime - start.tms_stime) / 60.0;
            ingrestime = (stop.tms_cutime - start.tms_cutime
                      + stop.tms_cstime - start.tms_cstime) / 60.0;
            cputime = equeltime + ingrestime - CPUOHEAD;
            etime = estop - estart - ELOHEAD;
            printf("\ncell %s, top level\n", cname);
            printf("cpu time:  equel = %3.1f", equeltime);
            printf("  ingres = %3.1f", ingrestime);
            printf("  o'head constant = %3.1f", CPUOHEAD);
            printf("  net = %3.1f\n", cputime);
            printf("elapsed time = %ld  o'head constant = %ld  net = %ld\n",
                      estop - estart, ELOHEAD, etime);
            printf("queries = %d\n", queries);
            printf("geometry tuples = %ld\n", tuples);

            printf("tuples per cpu second = %3.1f", tuples / cputime);
            if (tuples > 0)
                      printf("  cpu msec per tuple = %2.0f\n",
                                (cputime * 1000) / tuples);
```

```
        else
                printf("\n");

        printf("tuples per elapsed second = %3.1f", (float)tuples / etime);
        if (tuples > 0)
                printf("   elapsed msec per tuple = %2.0f\n",
                        ((float)etime * 1000) / tuples);
        else
                printf("\n");
}
```

Test 2: Design Tree Expansion with Geometry Retrieval

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include "macros.h"

#define CPUOHEAD 0.6
#define ELOHEAD 1

main(argc, argv)
int argc;
char *argv[];
{
##          char *cname, *dbname, *defstatus;
##          long cellid;

            dbname = *++argv;
            --argc;
            printf("\nProgram %s", "@(#)expand.q                1.14 2/11/83");
            printf("   database %s\n", dbname);
            while (--argc > 0)
            {
                    cname = *++argv;
                    cellid = NULL;
                    defstatus = "n";
##                  ingres dbname
##                  range of c is cell_master
##                  retrieve (cellid = c.master_id, defstatus = c.defined)
##                  where c.name = cname
##                  {
                                ;
##                  }
##                  exit
                    if (cellid == NULL || *defstatus != 'y')
                    {
                            printf("\n%s not found or undefined.\n", cname);
                    }
                    else
                    {
                            printf("\ncell %s, id is %ld\n", cname, cellid);
                            expand(dbname, cname, cellid);
                    }
            }
            printf("\n");
```

```
}

expand(dbname, cname, cellid)
char *dbname, *cname;
long cellid;
{
##          long ecellid;
##          int egener;
##          char *elname[6];
##          float ex1, ex2, ey1, ey2;
##          int i;
            int toggle, queries;
            long tuples, cells;
            extern int IItupcnt;
            struct tms start, stop;
            long time(), estart, estop, etime;
            float equeltime, ingrestime, cputime;

            ecellid = cellid;
            elname[0] = "NM";
            elname[1] = "NP";
            elname[2] = "ND";
            elname[3] = "NC";
            elname[4] = "NI";
            elname[5] = "NG";

            /* start the clock and expand the tree */

            times(&start);
            estart = time(0);
            tuples = 0;
            queries = 0;
##          ingres dbname
##          range of r is cell_ref
##          range of b is box
##          range of w is wire
##          range of p is polygon
##          create temp1 (cell=i4, gener=i2,
##                  t11=f4, t12=f4, t21=f4, t22=f4, t31=f4, t32=f4)
##          create temp2 (cell=i4, gener=i2,
##                  t11=f4, t12=f4, t21=f4, t22=f4, t31=f4, t32=f4)
##          create btemp (layer = c4, x1 = f4, y1 = f4, x2 = f4, y2 = f4)
##          create wtemp (layer = c4, x1 = f4, y1 = f4, x2 = f4, y2 = f4)
##          create ptemp (layer = c4, id = i4, vnum = i2, x = f4, y = f4)
##          range of bt is btemp
##        range of wt is wtemp
##        range of pt is ptemp
##        append to templ (cell=ecellid, gener=0, t11=1.0, t22=1.0)
##        range of t is templ
          queries += 6;
          toggle = 1;
          cells = 1;
          egener = 0;
          printf("Generation 0, 1 cell instance\n");
          do {

                    /* collect geometries of each type, applying transform. */
                    /* then retrieve one layer at a time for grouping. */

                    printf("collecting boxes..\n");
##                  append to btemp (layer = b.use,
##                            x1 = b.x1 * t.t11 + b.y1 * t.t21 + t.t31,
##                            y1 = b.x1 * t.t12 + b.y1 * t.t22 + t.t32,
##                            x2 = b.x2 * t.t11 + b.y2 * t.t21 + t.t31,
##                            y2 = b.x2 * t.t12 + b.y2 * t.t22 + t.t32)
##                  where b.owner = t.cell
                    queries++;

                    for (i = 0; i < 6; i++)
                    {
##                            retrieve (ex1 = bt.x1, ey1 = bt.y1,
##                                        ex2 = bt.x2, ey2 = bt.y2)
##                            where bt.layer = elname[i]
##                            {
                                        ;
##                            }
                              queries++;
                              tuples += IItupcnt;
                              printf("          %s layer, %d tuples\n",
                                        elname[i], IItupcnt);
                    }
##                  modify btemp to truncated
                    queries++;

                    printf("collecting wires..\n");
##                  append to wtemp (layer = w.use,
##                            x1 = w.x1 * t.t11 + w.y1 * t.t21 + t.t31,
##                            y1 = w.x1 * t.t12 + w.y1 * t.t22 + t.t32,
##                            x2 = w.x2 * t.t11 + w.y2 * t.t21 + t.t31,
##                            y2 = w.x2 * t.t12 + w.y2 * t.t22 + t.t32 )
##                  where w.owner = t.cell
                    queries++;

                    for (i = 0; i < 6; i++)
                    {
##                          retrieve (ex1 = wt.x1, ey1 = wt.y1,
##                                   ex2 = wt.x2, ey2 = wt.y2)
##                          where wt.layer = elname[i]
##                          {
                                    ;
##                          }
                            queries++;
                            tuples += IItupcnt;
                            printf("                %s layer, %d tuples\n",
                                    elname[i], IItupcnt);
                    }
##                  modify wtemp to truncated
                    queries++;

                    printf("collecting polygons..\n");
##                  append to ptemp (layer = p.use,
##                          id = p.polygon_id, vnum = p.vertnum,
##                          x = p.x * t.t11 + p.y * t.t21 + t.t31,
##                          y = p.x * t.t12 + p.y * t.t22 + t.t32)
##                  where p.owner = t.cell
                    queries++;
##                  modify ptemp to heapsort on layer, id, vnum
##                  retrieve (ex1 = pt.x, ey1 = pt.y)
##                  {
                            ;
##                  }
                    queries += 2;
                    tuples += IItupcnt;
                    printf("                %d tuples sorted & retrieved.\n", IItupcnt);
##                  modify ptemp to truncated
                    queries++;

                    /* print statistics for operation so far */

                    printf("\ncell %s so far: %ld cells, %ld geometry tuples\n",
                            cname, cells, tuples);
                    printf("queries = %d\n", queries);
                    printf("geometry tuples = %d\n", tuples);
```

```
                /* collect next generation in other temporary */
                /* compute new transform matrices */

                egener++;
                if (toggle==1)
                {
                        printf("\ngeneration %d, in temp2..\n", egener);
##                      append to temp2     (cell = r.child, gener = egener,
##                              t11 = t.t11 * r.t11 + t.t12 * r.t21,
##                              t12 = t.t11 * r.t12 + t.t12 * r.t22,
##                              t21 = t.t21 * r.t11 + t.t22 * r.t21,
##                              t22 = t.t21 * r.t12 + t.t22 * r.t22,
##                              t31 = t.t31 * r.t11 + t.t32 * r.t21 + r.t31,
##                              t32 = t.t31 * r.t12 + t.t32 * r.t22 + r.t32)
##                      where t.gener = egener - 1
##                      and t.cell = r.parent
                        queries++;
                        printf("                %d cell instances\n", IItupcnt);
                        cells += IItupcnt;
                        if (IItupcnt <= 0) break;
##                      modify temp1 to truncated
##                      range of t is temp2
                        queries++;
                        toggle = 2;
                } else
                {
                        printf("\ngeneration %d, in temp1..\n", egener);
##                      append to temp1     (cell = r.child, gener = egener,
##                              t11 = t.t11 * r.t11 + t.t12 * r.t21,
##                              t12 = t.t11 * r.t12 + t.t12 * r.t22,
##                              t21 = t.t21 * r.t11 + t.t22 * r.t21,
##                              t22 = t.t21 * r.t12 + t.t22 * r.t22,
##                              t31 = t.t31 * r.t11 + t.t32 * r.t21 + r.t31,
##                              t32 = t.t31 * r.t12 + t.t32 * r.t22 + r.t32)
##                      where t.gener = egener - 1
##                      and t.cell = r.parent
                        queries++;
                        printf("                %d cell instances\n", IItupcnt);
                        cells += IItupcnt;
                        if (IItupcnt <= 0) break;
##                      modify temp2 to truncated
##                      range of t is temp1
                        queries++;
                        toggle = 1;
                }
```

```
            } while (egener < 30);          /* exit if running too long */
##          destroy templ
##          destroy temp2
##          destroy btemp
##          destroy wtemp
##          destroy ptemp
##          exit
            queries += 5;

            /* stop the clock and print final statistics */

            times(&stop);
            estop = time(0);
            equeltime = (stop.tms_utime - start.tms_utime
                      + stop.tms_stime - start.tms_stime) / 60.0;
            ingrestime = (stop.tms_cutime - start.tms_cutime
                      + stop.tms_cstime - start.tms_cstime) / 60.0;
            cputime = equeltime + ingrestime - CPUOHEAD;
            etime = estop - estart - ELOHEAD;
            printf("\ncell %s, expanded\n", cname);
            printf("cpu time:  equel = %3.1f", equeltime);
            printf("  ingres = %3.1f", ingrestime);
            printf("  o'head constant = %3.1f", CPUOHEAD);
            printf("  net = %3.1f\n", cputime);
            printf("elapsed time = %ld  o'head constant = %ld  net = %ld\n",
                      estop - estart, ELOHEAD, etime);
            printf("queries = %d\n", queries);
            printf("geometry tuples = %d\n", tuples);

            printf("tuples per cpu second = %3.1f", tuples / cputime);
            if (tuples > 0)
                      printf("  cpu msec per tuple = %2.0f\n",
                                (cputime * 1000) / tuples);
            else
                      printf("\n");

            printf("tuples per elapsed second = %3.1f", (float)tuples / etime);
            if (tuples > 0)
                      printf("  elapsed msec per tuple = %2.0f\n",
                                ((float)etime * 1000) / tuples);
            else
                      printf("\n");
}
```

Test 3: Geometry Retrieval in a Small Area

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include "macros.h"

#define CPUOHEAD 0.6
#define ELOHEAD 1

main(argc, argv)
int argc;
char *argv[];
{
##      char *cname, *dbname, *defstatus;
##      long cellid;

        dbname = *++argv;
        --argc;
        printf("\nProgram %s", "@(#)area.q                  1.7 2/11/83");
        printf("  database %s\n", dbname);
        while (--argc > 0)
        {
                cname = *++argv;
                cellid = NULL;
                defstatus = "n";
##              ingres dbname
##              range of c is cell_master
##              retrieve (cellid = c.master_id, defstatus = c.defined)
##              where c.name = cname
##              {
                        ;
##              }
##              exit
                if (cellid == NULL || *defstatus != 'y')
                {
                        printf("\n%s not found or undefined.\n", cname);
                }
                else
                {
                        printf("\ncell %s, id is %ld\n", cname, cellid);
                        getarea(dbname, cname, cellid);
                }
        }
        printf("\n");
```

```
}

getarea(dbname, cname, cellid)
char *dbname, *cname;
long cellid;
{
##        long ecellid;
##        char *elname[6];
##        float ex1, ex2, ey1, ey2;
##        int i;
##        long left, right, bottom, top;
##        long b1left, b2left, b1right, b2right;
##        long b1bottom, b2bottom, b1top, b2top;
          extern int IItupcnt;
          long width, height;
          int queries;
          long tuples;
          struct tms start, stop;
          float equeltime, ingrestime, cputime;
          long time(), estart, estop, etime;

          ecellid = cellid;
          elname[0] = "NM";
          elname[1] = "NP";
          elname[2] = "ND";
          elname[3] = "NC";
          elname[4] = "NI";
          elname[5] = "NG";

          /* find the middle ninth of the area covered by top-level boxes */
          /* (assume other geometry likely to be there too) */
##        ingres dbname
##        range of b is box
##        retrieve (b1left = min(b.x1 where b.owner = ecellid),
##                  b1right  = max(b.x1 where b.owner = ecellid),
##                  b1bottom = min(b.y1 where b.owner = ecellid),
##                  b1top    = max(b.y1 where b.owner = ecellid),
##                  b2left   = min(b.x2 where b.owner = ecellid),
##                  b2right  = max(b.x2 where b.owner = ecellid),
##                  b2bottom = min(b.y2 where b.owner = ecellid),
##                  b2top    = max(b.y2 where b.owner = ecellid))
##        {
                  ;
##        }
##        exit
```

```
          if ((left = b1left) > b2left)
                    left = b2left;
          if ((right = b1right) < b2right)
                    right = b2right;
          if ((bottom = b1bottom) > b2bottom)
                    bottom = b2bottom;
          if ((top = b1top) < b2top)
                    top = b2top;
          printf("area occupied by cell: x = %ld to %ld,", left, right);
          printf("\n                              y = %ld to %ld\n", bottom, top);

          width = right - left;
          height = top - bottom;

          left += width / 3;
          right -= width / 3;
          bottom += height / 3;
          top -= height / 3;
          printf("middle ninth: x = %ld to %ld,", left, right);
          printf("\n                    y = %ld to %ld\n", bottom, top);

          /* start the clock */

          times(&start);
          estart = time(0);
          tuples = 0;
          queries = 0;
##        ingres dbname
##        range of r is cell_ref
##        range of b is box
##        range of w is wire
##        range of p is polygon

          /* collect geometries of each type in a temporary relation, */
          /* then retrieve one layer at a time for grouping. */

          printf("collecting boxes..\n");
##        retrieve into btemp (layer = b.use,
##                  x1 = b.x1, y1 = b.y1, x2 = b.x2, y2 = b.y2)
##        where b.owner = ecellid
##        and (b.x1<=right or b.x2<=right) and (b.x1>=left or b.x2>=left)
##        and (b.y1<=top or b.y2<=top) and (b.y1>=bottom or b.y2>=bottom)
          queries++;

##        range of bt is btemp
```

```
        for (i = 0; i < 6; i++)
        {
##              retrieve (ex1 = bt.x1, ey1 = bt.y1, ex2 = bt.x2, ey2 = bt.y2)
##              where bt.layer = elname[i]
##              {
                        ;
##              }
                queries++;
                tuples += IItupcnt;
                printf("                %s layer, %d tuples\n", elname[i], IItupcnt);
        }
##      destroy btemp
        queries++;

        printf("collecting wires..\n");
##      retrieve into wtemp (layer = w.use,
##              x1 = w.x1, y1 = w.y1, x2 = w.x2, y2 = w.y2)
##      where w.owner = ecellid
##      and (w.x1<=right or w.x2<=right) and (w.x1>=left or w.x2>=left)
##      and (w.y1<=top or w.y2<=top) and (w.y1>=bottom or w.y2>=bottom)
        queries++;

##      range of wt is wtemp
        for (i = 0; i < 6; i++)
        {
##              retrieve (ex1 = wt.x1, ey1 = wt.y1, ex2 = wt.x2, ey2 = wt.y2)
##              where wt.layer = elname[i]
##              {
                        ;
##              }
                queries++;
                tuples += IItupcnt;
                printf("                %s layer, %d tuples\n", elname[i], IItupcnt);
        }
##      destroy wtemp
        queries++;

        printf("collecting polygons..\n");
##      retrieve into ptemp (layer = p.use, id = p.polygon_id,
##              vnum = p.vertnum, x = p.x, y = p.y)
##      where p.owner = ecellid
        queries++;
##      modify ptemp to heapsort on layer, id, vnum
##      range of pt is ptemp
##      retrieve (ex1 = pt.x, ey1 = pt.y)
##          where max(pt.x by pt.id) > left
##          and min(pt.x by pt.id) < right
##          and max(pt.y by pt.id) > bottom
##          and min(pt.y by pt.id) < top
##          {
                    ;
##          }
            queries += 2;
            tuples += IItupcnt;
            printf("            sorted, %d tuples retrieved.\n", IItupcnt);
##          destroy ptemp
##          exit
            queries++;

            /* stop the clock and print statistics */

            times(&stop);
            estop = time(0);
            equeltime = (stop.tms_utime - start.tms_utime
                    + stop.tms_stime - start.tms_stime) / 60.0;
            ingrestime = (stop.tms_cutime - start.tms_cutime
                    + stop.tms_cstime - start.tms_cstime) / 60.0;
            cputime = equeltime + ingrestime - CPUOHEAD;
            etime = estop - estart - ELOHEAD;
            printf("\ncell %s, top level geometry in middle ninth area\n", cname);
            printf("cpu time:  equel = %3.1f", equeltime);
            printf("  ingres = %3.1f", ingrestime);
            printf("  o'head constant = %3.1f", CPUOHEAD);
            printf("  net = %3.1f\n", cputime);
            printf("elapsed time = %ld  o'head constant = %ld  net = %ld\n",
                    estop - estart, ELOHEAD, etime);
            printf("queries = %d\n", queries);
            printf("geometry tuples = %ld\n", tuples);

            printf("tuples per cpu second = %3.1f", tuples / cputime);
            if (tuples > 0)
                    printf("  cpu msec per tuple = %1.0f\n",
                            (cputime * 1000) / tuples);
            else
                    printf("\n");

            printf("tuples per elapsed second = %3.1f", (float)tuples / etime);
            if (tuples > 0)
                    printf("  elapsed msec per tuple = %1.0f\n",
                            ((float)etime * 1000) / tuples);
```

```
                else
                        printf("\n");
}
```

APPENDIX B

Tree Expansion Tests of Chapter 3

Table 1
Expansion of a Tree Without Duplicate Tuples
Data: RISC CPU

Test Number	Algorithm	Direct Update	Result Struct	Global Dup Elim on Result	Global Dup Elim During Walk	Dup Elim Within Gener.	Dup Elim on Path to Root	Tuples in Result	Disk Reads	Disk Writes	Disk Time @ 30 msec	CPU Sec	Elapsed Time
1	Breadth-First		Heap					14050	1247	1831	92	301	333
2	Breadth-First	Yes	Heap					14050	1227	909	64	314	278
3	Breadth-First		Hash	Yes	Yes	Yes		14050	5560	4003	287	568	733
4	Breadth-First		Heap			Yes		14050	1239	2745	120	491	551
5	Depth-First		Heap					14050	805	907	51	257	281
6	Depth-First	Yes	Heap					14050	814	907	52	250	264
7	Depth-First		Hash	Yes				14050	4667	3380	241	523	678
8	Depth-First	Yes	Hash	Yes	Yes			14050	7564	4149	351	532	747
9	Depth-First		Heap				Yes	14050	805	907	51	259	283
10	Depth-First	Yes	Heap				Yes	14050	814	907	52	250	263
11	Depth-First		Hash	Yes			Yes	14050	4667	3380	241	522	675
12	Depth-First	Yes	Hash	Yes	Yes		Yes	14050	7564	4149	351	533	747
13	Breadth-First		None						1238	915	65	280	308
14	Breadth-First		None			Yes			1230	1829	92	477	532
15	Depth-First		None						804	0	24	225	236
16	Depth-First		None				Yes		804	0	24	231	241

Table 2
Expansion of a Tree Without Duplicate Tuples.
Data: RISC Cache

Test Number	Algorithm	Direct Update	Result Struct	Global Dup Elim on Result	Global Dup Elim During Walk	Dup Elim Within Gener.	Dup Elim on Path to Root	Tuples in Result	Disk Reads	Disk Writes	Disk Time @ 30 msec	CPU Sec	Elapsed Time
17	Breadth-First		Heap					9854	2839	1279	124	267	301
18	Breadth-First	Yes	Heap					9854	2830	636	104	265	301
19	Breadth-First		Hash	Yes	Yes	Yes		9854	5149	2294	223	463	575
20	Breadth-First		Heap			Yes		9854	2843	1919	143	389	435
21	Depth-First		Heap					9854	2517	635	95	239	266
22	Depth-First	Yes	Heap					9854	2521	635	95	238	256
23	Depth-First		Hash	Yes				9854	3874	1658	166	428	508
24	Depth-First	Yes	Hash	Yes	Yes			9854	3936	1659	168	432	531
25	Depth-First		Heap				Yes	9854	2517	635	95	242	269
26	Depth-First	Yes	Heap				Yes	9854	2521	635	95	239	258
27	Depth-First		Hash	Yes			Yes	9854	3874	1658	166	428	507
28	Depth-First	Yes	Hash	Yes	Yes		Yes	9854	3936	1659	168	428	526
29	Breadth-First		None						2836	641	104	258	286
30	Breadth-First		None			Yes			2840	1281	124	382	422
31	Depth-First		None						2516	0	75	219	234
32	Depth-First		None				Yes		2516	0	75	221	237

Table 3
Expansion of a Tree Having Many Duplicate Tuples
Data: RISC CPU

Test Number	Algorithm	Direct Update	Result Struct	Global Dup Elim on Result	Global Dup Elim During Walk	Dup Elim Within Gener.	Dup Elim on Path to Root	Tuples in Result	Disk Reads	Disk Writes	Disk Time @ 30 msec	CPU Sec	Elapsed Time
33	Breadth-First		Heap					14050	878	353	37	161	179
34	Breadth-First	Yes	Heap					14050	857	169	31	137	160
35	Breadth-First		Hash	Yes	Yes	Yes		259	438	70	15	27	43
36	Breadth-First		Heap			Yes		2881	410	115	16	33	57
37	Depth-First		Heap					14050	805	167	29	123	138
38	Depth-First	Yes	Heap					14050	813	167	29	121	134
39	Depth-First		Hash	Yes				259	815	10	25	132	143
40	Depth-First	Yes	Hash	Yes	Yes			259	493	109	18	20	33
41	Depth-First		Heap				Yes	14050	805	167	29	123	137
42	Depth-First	Yes	Heap				Yes	14050	813	167	29	120	133
43	Depth-First		Hash	Yes			Yes	259	815	10	25	133	144
44	Depth-First	Yes	Hash	Yes	Yes		Yes	259	493	109	18	21	33
45	Breadth-First		None						869	177	31	143	161
46	Breadth-First		None			Yes			401	71	14	30	52
47	Depth-First		None						804	0	24	102	116
48	Depth-First		None				Yes		804	0	24	101	114

Table 4
Expansion of a Tree Having Many Duplicate Tuples
Data: RISC Cache

Test Number	Algorithm	Direct Update	Result Struct	Global Dup Elim on Result	Global Dup Elim During Walk	Dup Elim Within Gener.	Dup Elim on Path to Root	Tuples in Result	Disk Reads	Disk Writes	Disk Time @ 30 msec	CPU Sec	Elapsed Time
49	Breadth-First		Heap					9854	2581	245	85	173	188
50	Breadth-First	Yes	Heap					9854	2571	118	81	162	210
51	Breadth-First		Hash	Yes	Yes	Yes		113	383	35	13	27	37
52	Breadth-First		Heap			Yes		3763	359	107	14	34	50
53	Depth-First		Heap					9854	2517	117	79	149	168
54	Depth-First	Yes	Heap					9854	2520	117	79	147	163
55	Depth-First		Hash	Yes				113	2527	10	76	152	165
56	Depth-First	Yes	Hash	Yes	Yes			113	394	51	13	23	31
57	Depth-First		Heap				Yes	9854	2517	117	79	147	167
58	Depth-First	Yes	Heap				Yes	9854	2520	117	79	147	164
59	Depth-First		Hash	Yes			Yes	113	2527	10	76	152	165
60	Depth-First	Yes	Hash	Yes	Yes		Yes	113	394	51	13	24	32
61	Breadth-First		None						2578	125	81	164	178
62	Breadth-First		None			Yes			356	59	12	30	44
63	Depth-First		None						2516	0	75	133	152
64	Depth-First		None				Yes		2516	0	75	133	151