AN EXPERIMENTAL DISK-RESIDENT SPATIAL INFORMATION SYSTEM

by

Stephen Michael Choquette

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

L.G. Shapiro, Chairman

R.M. Haralick

E.A. Fox

December, 1985

Blacksburg, Virginia

ACKNOWLEDGEMENT

First and foremost, I'd like to thank my wife who provided support and understanding, even when it looked like this project would go on forever. Dr. Linda Shapiro, my committee chairperson, also deserves my heartfelt thanks since she provided the inspiration for the spatial database project. I also gratefully acknowledge the support of and who set the standards which I struggled to maintain in my disk-resident database system.

Finally, I'd like to thank my employer, , which provided much needed financial encouragement throughout the long tenure of this project.

CONTENTS

CHAPTER 1 - INTRODUCTION

Background

In a Master's thesis by Vaidya [49], an experimental geographical information system was introduced which used a formal organizational structure called a spatial data structure (SDS) as its basic building block. An SDS, as defined by Shapiro and Haralick in [37], is a set of relations that can be used to represent a geographic entity. The set of relations that comprise an SDS may contain tuples which refer to other spatial data structures, thus providing a recursive relational structure that is flexible enough to represent low-level geographic entities such as a single point, to high-level entities such as states and counties.

Refer to [49] for a survey of literature concerning geographic information systems, as well as additional information explaining how spatial data structures can be used to represent geographic entities.

Outline of Project

The geographic system developed in [49] was primarily a memory-resident system. Secondary storage devices such as disks were only used to store the database when it was not in use. This thesis describes the implementation of a similar geographic information system that was designed to reside entirely on disk. Like the system described in [49], the disk-resident system will be based on the stack-oriented query

language interpreter described by Minden in [30]. Unlike the memory-resident system, however, the disk-resident system was designed to serve a larger database community - one that is concerned with representing non-geographic entities such as traditional student-grades relationships.

To accomplish this goal, the concept of a prototype was extended. In [49], a prototype described basic geographic entities such as regions, roads, and labels. For example, Virginia and North Carolina would be represented by SDSs of type STATE, and should both be built according to the STATE prototype. Unfortunately, the system described in [49] had no database commands to describe what SDSs of type "STATE" should look like. The database user was required to know how relations and SDSs were implemented (e.g. tree pointers, catalog headers). With this information, a clever user could ensure that two STATE spatial data structures consisted of similar relations and fields.

In the disk-resident system, interpreter commands were provided to allow a database user to define relational and SDS prototypes, which identified the structure and composition of relations and spatial data structures. These prototypes provided three major advantages:

- Consistency across similar relations and SDSs
- Simplicity in creating relations and SDSs
- Integrity in ensuring that prototype field qualifications were met.

The prototypes also produced one additional benefit: the user interface was simplified since the database user no longer had to be concerned with the underlying implementation.

Outline of Thesis

In Chapter 2, several other relational database systems will be reviewed. The primary goal of each analysis will be to identify the logical and physical approaches to data representation, as well as the user interface.

Chapters 3 and 4 identify the physical and logical structure of the experimental disk-resident spatial information system developed for this project. As will be seen, this experimental system combines features of existing relational databases with new approaches to represent and manipulate information in a database.

The Query Language Interpreter will be presented in Chapter 5. Through the interpreter, a database user can issue a variety of commands to perform database operations, as well as create sophisticated program control structures.

Chapters 6-8 discuss the security, integrity, and recovery aspects of the disk-resident system.

During the development of this project, various implementation problems were encountered. These problems and solutions are presented in Chapter 9.

Chapter 10 discusses the performance of the disk-resident system. Statistics are presented comparing how the database commands performed when run under a variety of test environments.

Chapter 11 uses the results from earlier chapters to draw conclusions concerning the disk-resident system and presents some directions for future work.

Following the bibliography are related appendices that illustrate the various types of files recognized by the query language interpreter.

CHAPTER 2 - SURVEY OF RELATED LITERATURE

In this chapter, the logical and physical structures of several existing relational database systems will be described. By no means is this a a comprehensive list of all existing relational database management systems. Instead, the goal of this survey is to describe several different approaches to implementing a relational database system.

"Bibliography" on page 315 provides a list of literature concerning relational database management systems. [36] is a good starting point since it compares the logical structures of 14 existing relational database management systems.

The level of detail provided for each relational database management system depends on the quantity and quality of the reference information available. Many sources, for example, did not delve into the underlying physical structure of their database systems, concentrating primarily on query optimization.

The relational database systems described herin should not be considered static. Many of the systems described have evolved significantly (notably MDB) and additional research should be performed to determine the latest status of these systems.

DAMAS

Overview

DAMAS, described in [34-35], is an experimental relational data management system developed at MIT to test a new approach to solving the problem of data independence (i.e. divorcing the database user from knowledge of the underlying storage structures and search algorithms).

DAMAS is divided into two levels. At the lowest level, multiple storage modules are aware of the physical representation of relations and respond to certain types of requests from the higher level component, the Multi-Tuple Variable Module (MTVM). The mission of the MTVM is to evaluate the user query and decompose it into an expression of one-tuple variable quantifications called Primitive Boolean Conditions (PBCs). Appropriate storage modules are then called to evaluate the PBCs. Figure 2.1 shows the relationship between the two DAMAS levels.

User Interface

The view of the database seen by a user is as a set of named relations. Each relation is a set of tuples, stored in a file. Being a set, no two tuples in a relation may be identical. Relations in DAMAS are logically tables, with each row being a tuple, and each column being an attribute.

Users interact with a DAMAS database in a MULTICS environment using a query language similar to DSL/ALPHA. The query language permits the use of any number of tuple variables, each of which can be unquantified, existentially quantified, or universally quantified.

Logical Structure

The mission of the MTVM is to decompose the user query into one or more PBCs and to call Storage Modules to perform certain operations on the stored relations, including

- returning a sequence of tuples satisfying some condition,
- determining whether a relation has any tuples which satisfy specific qualifications, and
- eliminating from further consideration in the current computation all tuples which satisfy some input condition.

Successive interations, interspersed with calls to Storage Modules to obtain information such as the expected number of tuples in a PBC subset, refine the original PBC into one that most efficiently operates on the stored relations.

Physical Structure

The Storage Modules appear to the MTVM as a set of primitives that can be used to operate on relations, given a PBC to identify the tuples of interest. For example, the MTVM may request that Storage Module 1 restrict its attention to the subset identified by PBC1, for all subsequent operations on a specific tuple variable. The Storage Module, then, is responsible for accessing the relation and only returning the tuples to the MTVM which meet the qualifying condition, PBC1.

Each Storage Module handles <u>one</u> particular means of storing relations (e.g. multi-list files, hashing, B-trees, inverted files). The system

designer assigns a relation to a specific Storage Module based on which storage structure will provide the most efficient organization for the type of data being storaged. DAMAS was designed to accept an unlimited number of Storage Modules. This was accomplished by requiring that each Storage Module have the same interface with the MTVM.

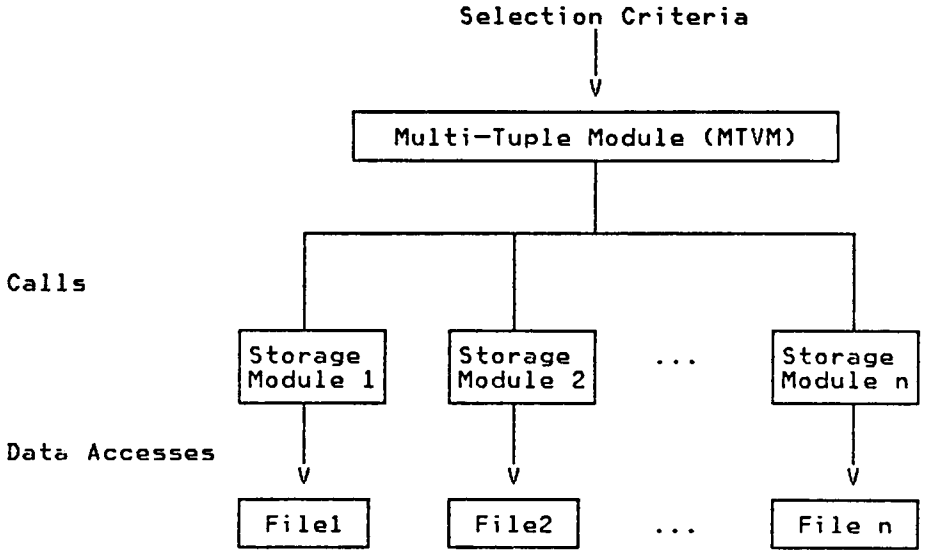


Figure 2.1 - Organization of DAMAS

Database 2

Overview

Database 2 (DB2) is a relational database management system developed by IBM based on preliminary work with System R [1-4], [18], [36], PRTV [10], [33], [36], IMS/VS, SQL/DS [40], and new technology. Running in an MVS environment, DB2 allows concurrent access to databases by IMS/VS-DC, CICS, and TSO users, both interactive and batch.

User Interface

Data are represented in the form of tables. Each row in the table is divided into a set of uniquely labeled columns, ordered from left to right. A base table is an autonomous, named table, unlike views which are not stored, but are derived from one or more base tables. Users may define one or more indexes on a base table. Each index resides in its own dataset, separate from the table data, and is maintained current with the data, thus providing fast access for queries that are predicated by the index or that request data to be ordered by a specific index. Refer to "Physical Structure" on page 13 for more information concerning the major storage structures used by DB2.

Most of the function of DB2 is available through the DB2 Interactive Interface (DB2I). THis interface provides the ability to execute SQL statements, invoke prewritten application programs, issue operator

commands, invoke utilities, and prepare application programs for execution.

Logical Structure

DB2 is composed of 6 major components, as is shown in Figure 2.2.

- Subsystem Services (SSS),
- The Relational Data Services (RDS),
- The Data Manager (DM),
- The Buffer Manager (BM),
- The IMS Resource Lock Manager (IRLM), and
- The Log Manager (LM).

The higher level components will be discussed in this section, while the description of the Buffer, Lock, and Log Managers will be deferred to the overview of DB2's physical structure.

Subsystem Services (SSS)

SSS regulates the flow of application requests into RDS. It maintains a record of each application currently executing code within a DB2 address space, and ensures that a request to stop DB2 is held until all existing work is terminated. SSS also verifies that the requesting application is authorized to use a particular thread through DB2. Finally, SSS routes the application's request to RDS for processing.

Relational Data Services (RDS)

RDS is responsible for materializing the external views of data from stored data. RDS chooses the access path method for evaluating queries and then invokes the Data Manager to retrieve or alter stored data. RDS

will order data using either an internal sort, or an index, if required (e.g. by a JOIN operation).

Data Manager (DM)

DM manages all stored data by providing access to data and by using a sequential scan or index, as specified by RDS. DM provides hash access to the DB2 catalog and other system tables, and link access for records tied together in multiple tables. The Data Manager also manages concurrency by calling the IRLM to obtain and release locks.

Authorization

The creator of a table may identify a set of users who may access the table for retrieval or update. By granting access to a view, access may be restricted to a subset of the rows and columns of the table.

Concurrency and Locking

Applications can concurrently access and change data in a table. The changes will not be visible to another application until transaction disposition, at which time a decision is made to commit the changes or abort. Should another application desire to access changed data in an uncommitted state, that application must wait until transaction disposition.

A comprehensive range of locks can be acquired and released by DB2 using facilities of the IMS Resource Lock Manager (described later).

The View Mechanism

One or more views may be defined on a table or set of tables using the VIEW statement. Views are not supported by their own, physically separate, distinguishable stored data; instead, the the definition of a view in terms of other tables is stored in a catalog table called SYSVIEWS. This means that changes to the base relations manifest

themselves immediately in any views based on the relations. Views may be defined on top of other views.

Recovery

Three forms of recovery are provided:

- Units of Recovery - no changes are written to direct access storage until information has been written to a recovery log that permits DB2 to undo the changes.
- Emergency Restart - after a catastrophic error that caused DB2 to terminate abnormally, DB2 recognizes that it needs to recover up to the point of the error, minus all uncommitted Units of Recovery.
- Media Recovery - DB2 recovers from an external storage media error by using incremental image copies (backups) created by a DB2 utility service called Image Copy. When requested to by a user, the Recover utility consults the DB2 catalog, recognizes that a recovery is needed, and restores data using appropriate incremental image copies and the recovery log.

Physical Structure

The major storage structures of DB2 are user databases and several system databases. Each database is divided into a number of disjoint tablespaces and indexspaces, where a "space" is a dynamically extendable collection of pages. Each indexspace defines one index. Each space has associated with it a storage group, a collection of direct access volumes that will be used to retain the information in the space. Each tablespace is composed of one or more stored tables. Figure 2.3 (taken from [14]), presents a pictorial representation of the major DB2 storage structures.

A stored table is the physical represertation of a base table. The stored table describes a set of direct access files to be used for storage of rows of that table. Data are written in 4K blocks called pages, which correspond to the operating system page size.

Buffer Manager (BM)

The Buffer Manager manages the buffers that hold data and index pages and directs the VSAM Media Manager to transfer data and index pages between media storage and the buffers (virtual storage in the MVS environment). DB2 supports four distinct buffer pools. Three buffer pools hold 4K-byte pages, while one buffer pool holds 32K-byte pages. The 32K-byte pool allows DB2 to read and write large blocks of data with a single I/O operation. These larger blocks are used for table spaces that contain records longer than 4K-bytes. The DB2 installation is required to select a buffer pool when the tablespace or indexspace is created. This permits an installation to tune its database systems by allocating its databases among the various buffer pools.

A modified Least-Recently-Used algorithm is used to manage pages within the buffer pool. Updated pages are kept in the buffer pool in preference to nonupdated pages, when there are a sufficient number of buffers available. This takes into account the additional cost of paging out an updated page.

A database page is considered to be in use if the Data Manager has accessed the page, but not released it. Since the Data Manager may retain access to a page for a long time (e.g. to read and process multiple records on the page), the Buffer Manager will warn DM when it is running low on available buffers. At that point, the Data Manager will start acquiring and releasing each page every time it accesses a record. The Data Manager will be informed when sufficient buffers are available again. Pages are written only when all changes to the page

are committed or have been backed out, and when the log records that guarantee data consistency have been updated.

IMS Resource Locking Manager (IRLM)

The IRLM provides a variety of locks to minimize interference between concurrent users, and to prevent the users from accessing inconsistent (uncommitted) data. The DB2 user specifies the locking granularity for a table space when the table space is created. This allows DB2 to manage the acquiring and releasing of appropriate locks based on the desired level of data consistency specified for each transaction that accesses that table space.

Log Manager

The DB2 Log Manager records all modifications to data in the DB2 recovery log. When data is updated, the way the data looked before and after the modification is saved in the log, providing a means to undo an update. Refer to "Recovery" on page 13 for more information concerning the DB2 recovery process.

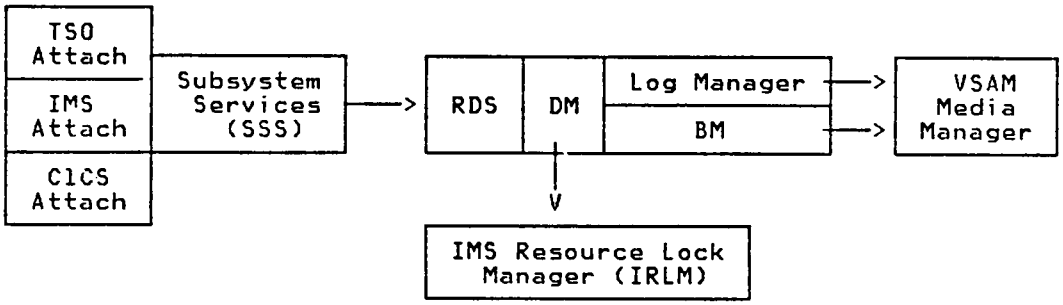


Figure 2.2 – Organization of DB2

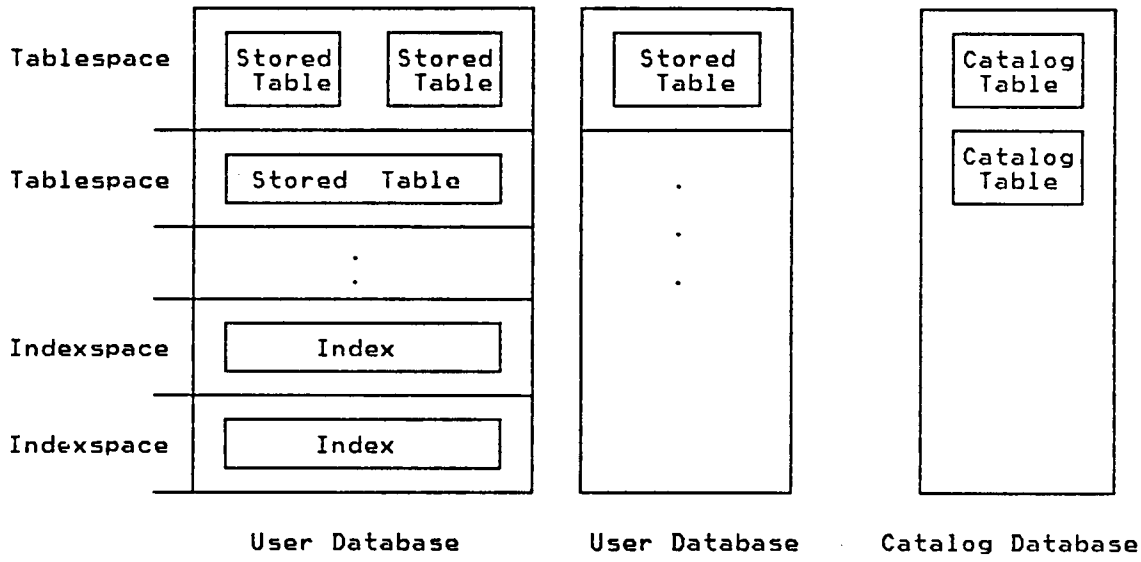


Figure 2.3 – DB2 – Major Storage Structures

Experimental Memory-Resident Spatial Information System

Overview

The experimental memory-resident spatial information system described in [49] was developed as a Master's thesis by Vaidya to test the feasibility of using a primitive building block called a spatial data structure to represent geographic information.

User Interface

The query language interpreter, first described by Minden in [30], is a stack-oriented language that provides the means through which a user can interact with a spatial database system. Using one or more commands in the query language, the user can perform any of the following function:

1. Access, create, delete, or modify individual tuples, relations, or spatial data structures in a selected database,
2. Perform arithmetic and boolean operations on stack entries,
3. Define variables, arrays, constants, and new commands to extend the query language,
4. Accept and direct interpreter input and output from an external VMS file,
5. Control the execution of the interpreter through the use of program control structures such as IF-THEN-ELSE, DO loops, and REPEAT-UNTIL loops, and
6. Perform a variety of debugging functions.

Logical Structure

The database system recognizes two types of relations: tree relations and list relations, and several atomic elements. The choice of which relation type to use depends on the desired format of the underlying structure used to store the relation.

Tree Relations

For tree relations, the first component of an N-tuple is stored on the first level, the second component on the second level, and so on.

The tree is structured so that all N-tuples in the relation with the same first component share the same node. All N-tuples with the same first two components share the same nodes on the first two level. Figure 2.4, taken from [49], shows how a relation consisting of 3 similar N-tuples would be stored.

N-tuples are always stored in lexigraphical order, facilitating searches for specific component values.

List Relations

List Relations store tuples in a list form, with the list of N-tuples growing downward, and the list of components of an N-tuple growing horizontally. The order of the N-tuples stored in a list relation is user-defined. Figure 2.5 shows how a list relation would be used to represent the three tuples presented in Figure 2.4.

Atomic Data Elements

The database system also recognizes 4 types of atomic data elements: integers, reals, character strings, and points. Integers and reals are represented using their PASCAL equivalents. Character strings are represented as one or more linked lists, with each list representing 15

characters of the total character string. Points are represented as integer pairs (X,Y).

Dictionaries

The database system maintains two dictionaries:

- The RDS Dictionary contains a header with the following information about each spatial data structure: 1) the SDS name, 2) a pointer to the SDS header, 3) a flag that the structure is in memory, 4) a flag indicating that the structure has changed and should be written back to disk, and 5) a pointer to a list of relations that compose this SDS.
- The REL Dictionary contains a header with the following information about each relation: 1) the relation name, 2) a pointer to the relation header, 3) a flag that structure is in memory, 4) a flag indicating that the structure has changed and should be written back to disk, 5) the type of relation (i.e. tree or list), and 6) a pointer to the first tuple in the relation.

Physical Structure

When a database is loaded, the RDS and REL dictionaries are read from disk into memory. As relations and spatial data structures are required, they are read in and the list or tree relations described earlier are created. To be able to selectively retrieve individual information from disk, each SDS and relation is stored in a separate file on disk. The names of the SDS and relations are used as the file names. An advantage of this approach is that VMS will create a new copy of a file (with a higher version number) when an SDS or relation is transferred back to disk.

The data within a file is stored sequentially. Each data element occupies one record, with records for the SDS or relation header preceding the actual tuple information. Using such an approach, the information on disk can be stored in a manner that parallels its eventual structure in memory.

Tuples: (a,b,c)
(a,b,d)
(b,c,d)

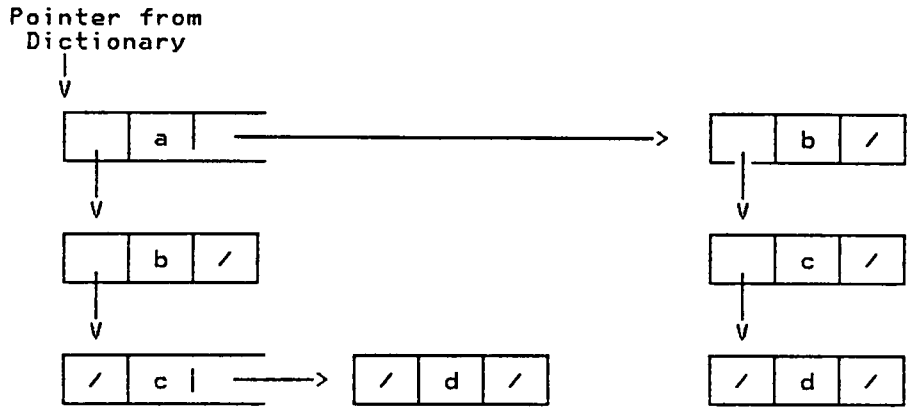


Figure 2.4 - Memory-Resident System - Representation of Tree Relations

Tuples: (a,b,c)
(a,b,d)
(b,c,d)

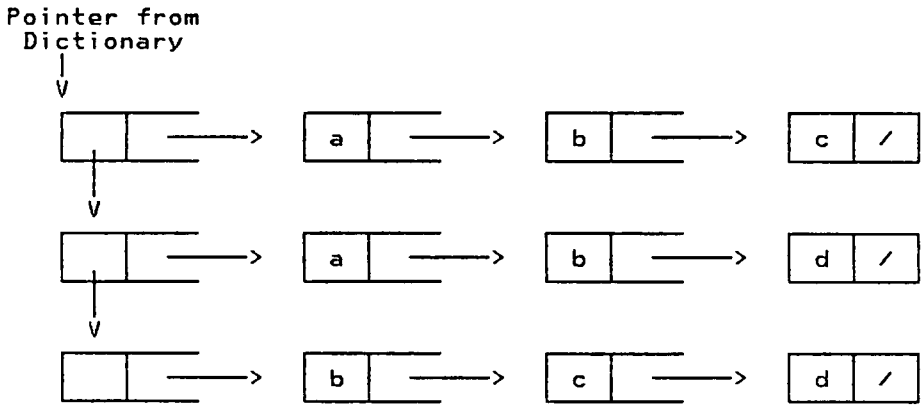


Figure 2.5 - Memory-Resident System - Representation of List Relations

INGRES

Overview

INGRES (Interactive Graphics and Retrieval System), described in [21-22], [40-43], and [36], is a relational database management system developed as a research project by the University of California at Berkeley.

The processes that support the INGRES database appear as standard user jobs to the UNIX operating system; this approach makes the INGRES database and its processing functions portable to other standard UNIX operating environments. One consequence of this decision is that UNIX performs all memory management for the database system.

The process structure created by issuing INGRES as a UNIX command is shown in Figure 2.6. In this structure, query processing is modularized into 4 isolated processes which regulate control and information flow through "pipes". Commands are passed only to the right, data and errors to the left. The process approach was chosen primarily to overcome PDP-11 address space limitations.

Process 1 is an interactive terminal monitor that permits a user to formulate, print, edit, and execute collections of INGRES commands. Process 2 parses each query and modifies it to realize integrity and view constraints. Process 3 optimizes queries and calls execution routines to perform the requested database commands. Process 4 supports utility commands and a deferred tuple update scheme.

When invoking INGRES from a C program, the Process 1 is eliminated since query statements imbedded in the C program were translated into appropriate C code and INGRES calls.

User Interface

User access to INGRES is either interactively via the UNIX command processor, or through a procedural language for batch use. Four interfaces are available:

- QUEL - Query language based on the relational calculus with facilities for data definition, retrieval, update, access control, and integrity verification.
- EQUEL - Similar in function to QUEL. Permits inclusion of QUEL statements in C programs.
- CUPID - Pictorial query language designed for the casual INGRES user.
- GEO-QUEL - Query language with the capability to present geographic data in map form.

Logical Structure

Process 1

Process 1 supports an interactive terminal monitor system that allows a user to interact with the INGRES database system through a calculus-based language called QUEL. QUEL supports commands to retrieve, append, replace, or delete tuples matching specified

qualifications, as well as utility commands to create and destroy databases and relations, bulk copy data, modify storage structures, and enforce database authorization and integrity constraints.

Process 2

Process 2 consists of routines to lexically analyze a user query, produce a query parse tree, and modify the parse tree to enforce authorization and integrity constraints.

The Authorization mechanism selectively restricts access of each database user by modifying a user query with protection predicates which restrict the user's access privileges. Each relevant protection predicate is grafted to the query parse tree, ANDing any protection predicates with those in the query. A parse tree for each restriction qualification is stored in the PROTECTION relation. File security is provided by the UNIX operating system.

The View Mechanism allows a user to define alternate windows on the database using the full query power of QUEL. Views are implemented as virtual relations in INGRES. Instead of physically storing a view as it does a relation, INGRES stores a partially processed description (a parse tree) of each view. The description defines the view in terms of its base restrictions. This means that when the base relation changes, all views against the relation change.

The Integrity Mechanism maintains certain constraints or consistency conditions on the data in the face of updates. Each integrity constraint is grafted to a parse tree for update requests, ANDing any predicate with those in the query ([40] and [17]). UNIX and QUEL commands provide for recovery from system failures and allow the creation of backup files.

Process 2 also supports concurrency control and maintains catalog relations for the database. The locking granularity of INGRES is at the relation level, but has the capability of working at the page level.

Physical Structure

Process 3

Process 3 decomposes user queries into single-variable queries which are then processed. Any tuple updates caused by the REPLACE, APPEND, or DELETE commands are spooled to a temporary file. Process 3 is aware of the database storage and file structures and calls access methods interface routines to perform the database commands.

Five storage structures are available to an INGRES user:

- Keyed Structures - the key is a tuple identifier (TID) and determines the page of the file on which the tuple resides. The TID consists of the page number, followed by an index into a line table (like System R offset slots).
- ISAM-Like Structures - fixed length tuples are distributed in an ISAM-like fashion, sorting the relation based on a particular key.
- Hash Structures - tuples are distributed on pages according to a key hashing function. Rapid access occurs when a specific tuple is to be retrieved.
- Compressed ISAM-Like and Hashed Structures - Compressed ISAM files remove blanks and portions of a tuple which match the preceding tuple. Compression is applied to each page independently and is used when the need for increased storage utilization outweighs the cost of coding and decoding tuples.

- Heap Structures - tuples are stored in a file in the order they were received, independent of the value or domain. The unique tuple identifier for a tuple is its byte offset within the file.

The query optimization process has three techniques to speed up the execution of a complex user query:

- Tuple substitution reduces a query to fewer tuple variables,
- One-variable detachment reduces the complexity of a query by limiting the range of the relation using restriction and projection,
- Reformatting attempts to replace a collection of complete sequential scans of a relation with a set of limited scans.

The One-variable processor then determines what key (if any) may be used profitably to access an individual relation, what value(s) of the key should be used in calls to the Access Method Interface system, and whether primary or secondary indexes should be used to access the relation. Tuples are retrieved according to the selected access strategy.

Process 4

Process 4 consists of routines that

- Invoke the system utility commands (e.g. modify relational structures and access methods),
- Process the tuple updates deferred from Process 3 (to avoid problems of order-dependent tuple updates), and
- Support system recovery in the event of a crash by using a temporary update file to "back out" incomplete updates.

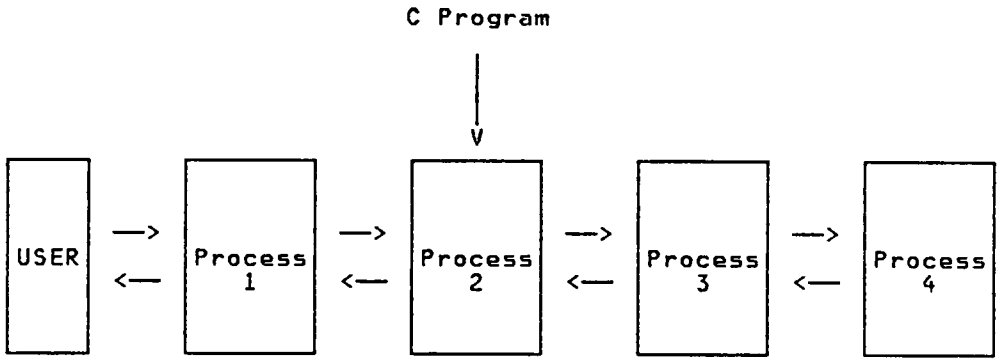


Figure 2.6 - Organization of INGRES

Great New Mini-Database

Overview

The Great New Mini-Database (MDB), described in [8], was initially developed in 1977 as a class project for CS5362 at Virginia Polytechnic Institute and State University. MDB is primarily a educational tool, representing the work of approximately 40 students over a period of 5 years.

MDB is a database management system for a relational database on a DEC VAX 11/780 with the VAX/VMS operating system. The MDB contains a variety of functions which enable the user to 1) create the initial database, 2) create, drop, and store relations, 3) query and update relations, and 4) insert and delete tuples in a relation.

User Interface

As is typical with most relational database systems, users perceive a relation as a simple table of data. Rows in the table, also called tuples, correspond to records in a file. Columns correspond to the attributes for which a user may store values.

The MDB system can handle up to 41 relations. Each relation may have from 1 to 36 attributes, which may be defined as indexed at the discretion of the user. Indexing an attribute makes inserting, deleting, and updating tuples slower, but increases the speed of

retrieval queries involving indexed attributes since B-trees are involved.

User access to MDB is via a SEQUEL-like command language. Twelve commands are provided to

- Create, load, store, and drop specified relations,
- Select desired values from tuples which fit a given predicate,
- Insert, delete, append, and update desired tuples, and
- Retrieve information about relations.

Logical Structure

The following data structures control most of the information about what is stored in the MDB database.

- The Database System Directory (DBSD) contains the name of each relation, the number of attributes in the relation, and the page number of the relation's directory. Up to 41 relations may be in the DBSD.
- The Relation Directory (RD) contains information about a relation in the database. There is a separate directory for each relation. The directory is one page long, permitting the specification of 36 attributes for each relation. Figure 2.7 illustrates the format of the MDB Relation Directory. (NOTE: all figures were taken from [8])
- The Tuple Storage Area Map (TSAMAP) provides an interface between the physical storage system of the VAX and the database. The TSAMAP keeps track of which physical pages are used by a relation and provides information about which pages are in memory, the number of

offset slots being used by page, the number of free words by page. Figure 2.8 shows the format of the TSAMAP.

- The Value/TID Storage Area (VALTID) contains keys which are attribute values. For each key, there is a list of TIDs for tuples having the key value. For example, if an indexed attribute is COLOR, the VALTID area may have an entry for BLUE, followed by a list of TIDs for tuples having COLOR=BLUE. Each VALTID page is a leaf of a B-tree. Refer to Figure 2.10 for a pictorial representation of a VALTID storage area.
- The Page Allocation Table keeps track of what pages have been allocated.

B-trees

Tuples can be located quickly using indexed attributes. To do tnis, a B-tree is created for each indexed attribute. The relation's entry in the Relation Dictionary points to the root of a B-tree for each indexed attribute. Assbciated with each key in the B-tree node is a pointer to the VALTID page that contains the TIDs of tuples with attributes having this key or with values greater than this key, but less than the next key on a B-tree page. Pages with keys meeting specific predicates are found quickly. The B-tree is kept balanced because a flat tree reduces the time required to locate a specific key. Figure 2.9 shows the format of a B-tree page (i.e. a B-tree node). Figure 2.10 shows the format of a VALTID page which will serve as the B-tree leaf node.

Physical Structure

The database size is 2**20 words of disk storage divided into 4096 pages of 512 bytes (256 words) each. Upon creation, each relation uses 3 database pages for tables, plus 2n pages for indexing if n attributes

are indexed. As tuples are inserted, additional pages are added as needed for the tuple storage and the index. No more than 63 tuples are stored per page, regardless of tuple size. Stored tuple sizes range from 2 to 255 words. The stored tuple is made up of each attribute's value, preceded by a 1-word length field.

The database is contained in one large disk file, which contains system directories, indexes (B-trees), and stored tuples. The system requires that the first two pages of the database be allocated for system use. The first page is the page map, with up to 4096 bits to indicate whether a specific page is in use. Fixed system information is stored on the second page of the database (e.g. the number of bytes per page).

Tuples can be located by using a Tuple Identifier, TID, which consists of a page number and an offset slot number. The page number is relative (versus being a VAX page number) and points to a word in the TSAMAP which contains the corresponding VAX page number. Thus, there is a hierarchy to locate tuples within a relation:

Given a resource name --> DBSD --> RD --> TSAMAP --> VAX page

Tuples occur sequentially on a page, starting with tuple 1. The length of each tuple is in an offset slot, located at the bottom of a page. Offset slots are numbered backwards (in a manner similar to System R), so slot 1 is the last word on the page, slot 2 precedes slot 1, etc. Figure 2.11 provides a pictorial representation of a database page.

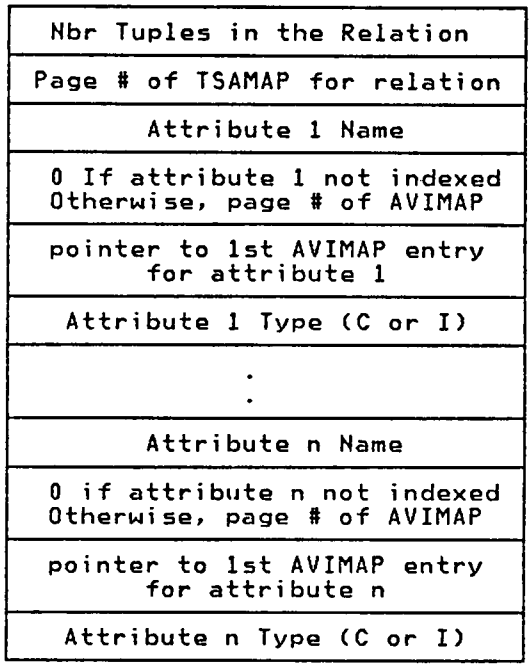


Figure 2.7 – MDB Relation Directory (RD)

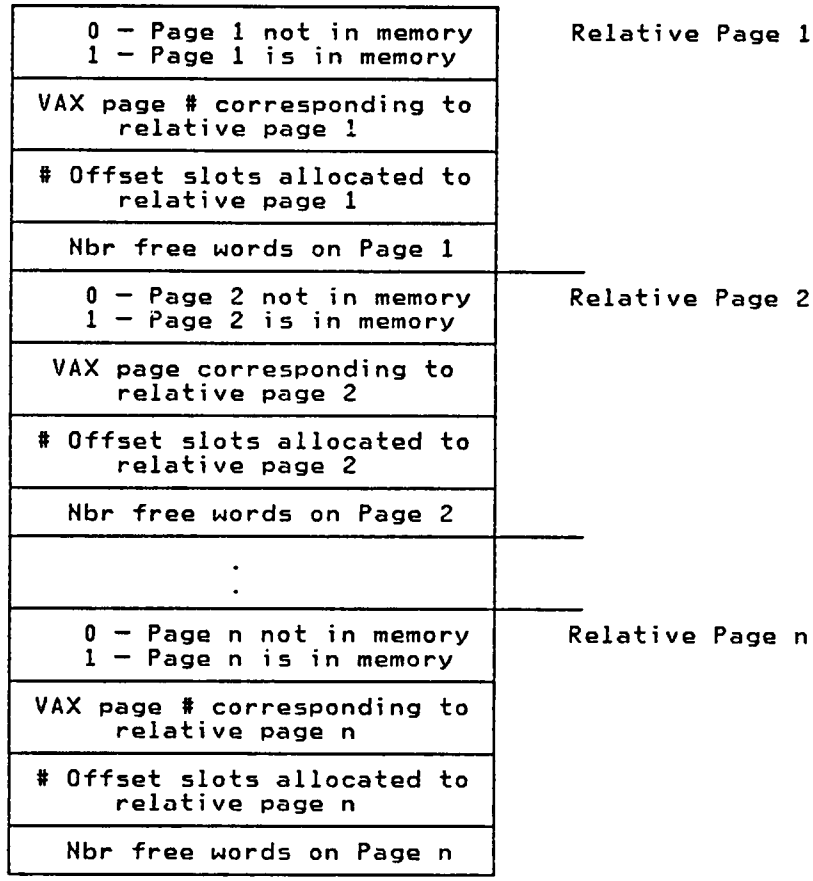


Figure 2.8 - Format of MDB Tuple Storage Area (TSAMAP)

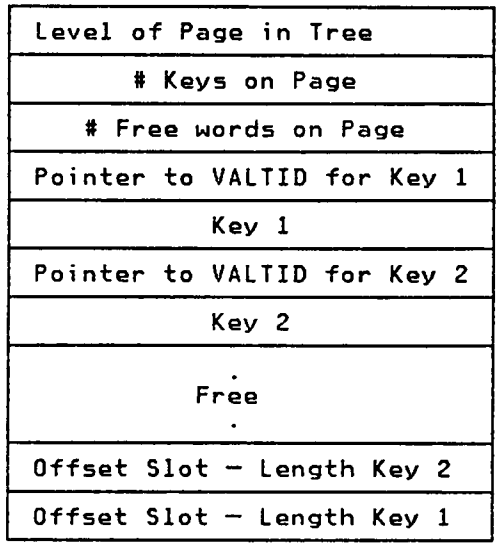


Figure 2.9 - MDB - Format of a B-tree Page

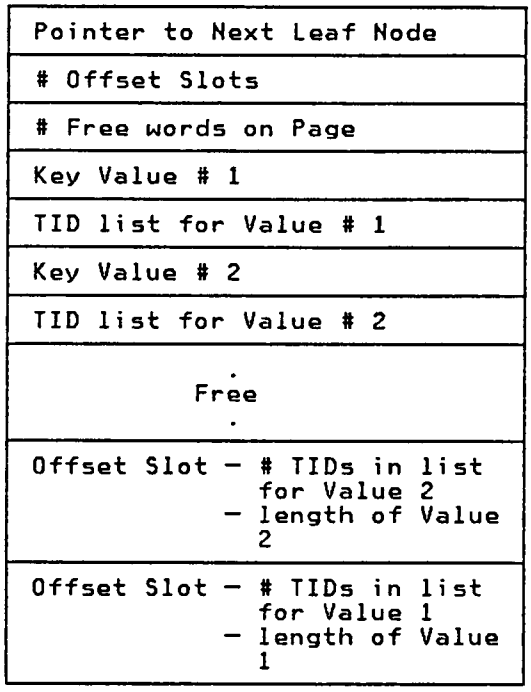


Figure 2.10 – MDB – Format of a B–tree Leaf Node Page (VALTID)

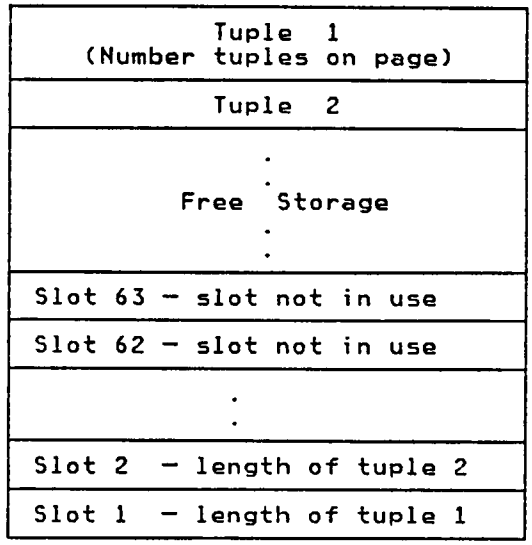


Figure 2.11 - MDB - Format of a Page Containing Tuples

Peterlee Relational Test Vehicle (PRTV)

Overview

The Peterlee Relational Test Vehicle (PRTV), discussed in [45-46] and [36], is an experimental relational database management system developed from 1970-1978 by the IBM United Kingdom Scientific Center to investigate the problems associated with designing and using high-level relational database systems for general applications. Based on experience with its prototype, the Peterlee IS/1 system (described in [32]), the PRTV design goals were to provide

1. a high-level, flexible database with functional extensibility,
2. a query language that allows major operations on large volumes of data to be performed using a single statement, and
3. an experimental system that could be used as either a stand-alone database, or as a data subsystem (relational front-end) for an applications system.

Since it was a research project, PRTV did not attempt to provide a complete database system. The PRTV project concluded in 1978 after being used in six major applications (mentioned in [36]). Knowledge obtained during the development of PRTV was incorporated in later IBM database systems such as System R and DB2.

PRTV is implemented as a two-level machine: a Top-End Subsystem, and a Bottom-End Subsystem. The Top-End is responsible for maintaining the interface with a PRTV user; it supports the query language and provides for general user extensions to handle non-standard relational operations. The Bottom-End consists of subroutines that handle records

representing tuples. The Bottom-End implements the relational operators, provides sorts, and permits tuple-at-a-time user extensions.

The interface between the two system levels is the Common Intermediate Language (CIL), a powerful and concise language that permits significant query optimization. The organization of PRTV is illustrated in Figure 2.12.

User Interface

The database user pictures a relation as an arbitrarily ordered set of tuples. No relation is permitted to have duplicate tuples. A database is a set of named relations. Names are partitioned into sets, representing those relations that can be seen by a particular database user.

User access to PRTV is through either a Top-End interpreter for the Information System Base Language (ISBL), or from previously linked user extension programs written in PL/I. Using ISBL operators, a user can manipulate the bulk data held in PRTV relations. Using extension programs, a user can perform any non-standard relational operations needed to satisfy the demands of his or her specific applications.

PRTV provides mechanisms to escape to user extension programs in order to handle non-standard relational operations. The extensions, composed of previously linked ISBL and PL/I commands, allow a user to tailor the data-entry and output formatting functions of database maintenance. Standard system extensions also exist to enter and list relations, and for performing basic arithmetic and string operations.

PRTV also implemented a concept of a workspace (WS), whereby a user could create and manipulate "temporary" copies of relations known to PRTV. All of the standard ISBL relational operators can be used to define and change these "temporary relations". ISBL commands also exist to permanently retain or destroy the relations.

The Relational File Interface

Relational files are the interface between a PRTV database and a user extension program. Using a relational file, the user program can transfer data to and from the database, one tuple at a time. The file concept is required because the standard ISBL operators only work on relations.

Logical Structure

The Top-End Subsystem

The Top-End Subsystem is similar to an interpretive compiler for the user query language. It analyzes query syntax, handles relation naming, checks the semantics of relational operations, and tries to choose the optimum data access path for a given set of user commands. Also embedded in the Top-End Subsystem is a mechanism that allows user-created PL/I programs to assist in database maintenance.

The output of the Top-End is a character string composed of identifiers and operators from the Common Intermediate Language (CIL), the interface language between the Top-End and Bottom-End Subsystems. The ordering of the operators and identifiers in the string corresponds to the optimum access path to the database for a given user query. The Bottom-End Subsystem recognizes each CIL operator and identifier in the string and performs the specified relational operations.

The Top-End Subsystem has the following components:

- The Syntax Analyzer evaluates incoming ISBL commands. Calls to user extensions are forwarded to the user function control routines, which ensure that the user program has been properly linked. Parameters are then translated and control is passed to the user function.
- Commands involving relational files are translated by Relational File Control into an equivalent operator in the Common Intermediate Language.
- The goal of the Optimizer is to select the best access path to the data stored by the Bottom-End Subsystem. Optimization is achieved by reorganizing and extending the tree for the CILstring that provides the interface with the Bottom-End. Operator nodes in the tree are relocated in order to execute the CIL operators in the fastest manner; the lower a node is in the tree, the sooner the operation will be executed. Commands involving bulk data are deferred, permitting optimization of an entire group of user commands, while giving the impression of immediate query response.

PRTV was developed with no provisions for concurrency and crash recovery. Security was provided by password protection at logon time, and guaranteeing that each user saw only his or her own set of relations, some of which may have been views held by other database users.

Concurrent use of a single PRTV database by many users is not supported, although different users can work with different parts of the same data. User-created relations are private, but can be shared explicitly.

The CIL Interface

Communication between the Top-End and Bottom-End Subsystems is through

the CILstrings produced by the ISBL interpreter. The CILstrings specify which relations and relational operators are needed to satisy a user request. Relations in a CILstring are identified as one of two basic data set types: bricks and streams. Bricks are physical sets of tuples. Streams represent the logical set of tuples created by performing a series of relational operations on one or more bricks. All writing by the Bottom-End is into bricks, while all reading is from streams. Both types of data sets appear as homogeneous sequential user files to the Bottom-End processes.

Physical Structure

The Bottom-End Subsystem

The Bottom-End Subsystem is a collection of subroutines that implement the CIL stream operators. Complex operations like union and storing large quantities of data are carried out at this level. The Bottom-End also provides for calls to tuple-at-a-time user extension programs.

The Bottom-End Subsystem operates on two types of data structures. Bricks are used to hold collections of tuples, while value sets are used to store character strings.

Bricks

Bricks are stored as sequential files on disk. Efficient storage is achieved by 1) standard fixed-length record formats, 2) blocking of data, 3) sorting, suppression, and compression of fields, and 4) the use of page indices.

Bricks are written in fixed-size pages using a page size chosen when the database is initially formatted. Records are blocked within a page in

order to reduce the number of disk accesses. The first record of each page is written in full at the head of the page. The remaining records are suppressed, compressed, and then written sequentially onto a page.

The page indices provide fast access for certain selection operations. The data pages of a brick are held in an array, along with the value of the first field in the first and last records of each page. Since all permanent bricks are kept sorted, page indices make the execution of stream operations more efficient.

Value Sets

One way the Bottom-End Subsystem achieves storage efficiency is in eliminating duplicate character data within each record. Instead, each distinct character string is stored in a separate area called a value set. Each value set has a unique identifier which is used in the brick to reference the value set.

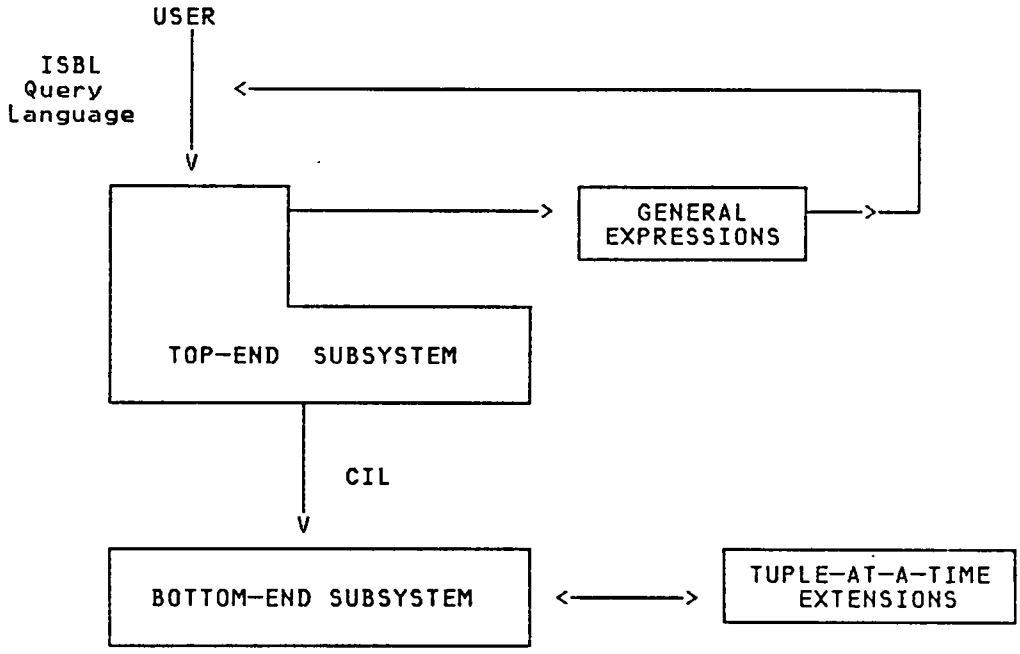


Figure 2.12 - Organization of PRTV

Query-By Example (QBE)

Overview

Query-By-Example (QBE) is an experimental relational database system developed as a research project by IBM's T.J. Watson Research Lab in Yorktown, N.Y. [36], [52-53], [55], and [12] all provide some information about QBE.

User Interface

A user views a QBE database as a collection of relations. A QBE relation is an ordered set of N-tuples whose components consist of elements from n-domains. Relations are represented by 2-dimensional tables in row/column form, with each column having a unique name. No duplicate tuples are allowed in a relation.

Users access a QBE database via either a tabular language or a linear systax language. The tabular language is interactive and permits a user at a terminal with a special screen editor to update the database by simply marking changes in specific columns of a pictorial 2-dimensional table. Queries are made in a similar manner, as is shown in Figure 2.13, (taken from [12]). The linear syntax language is the method through which a PL/1 or APL program can access the database.

Logical Structure

The QBE Dictionary is presented to the user as a collection of tables giving details about all tables and domains known to the system. The dictionary tables can be interrogated and updated using ordinary database commands. Each domain of a tuple may be declared to have any of the following properties:

1. An identification of whether the attribute is part of the key. The set of key attributes must uniquely identify a tuple within a relation.
2. A name for the domain of values for this attribute,
3. The data type (e.g. CHARACTER(5)), and
4. An indication of whether an index on the attribute is to be created and maintained.

Views

QBE permits the user to define static views of the database. These views are snapshots that can be queried or updated. Once stored, however, these snapshots are independent of the originating tables. As such, a snapshot is static (versus derived).

Integrity

QBE supports a limited form of domain support. Support is provide for primary keys, which are required and must be unique. Integrity constraints are inserted in the QBE dictionary by filling entries in blank tables (I.CONSTR rows). Figure 2.14 shows two integrity constraints: 1) Field F3 must be less than Field F2, and 2) Field F1 must be greater than 0 and Field F4 must be either "LONDON" or "PARIS".

The value in the parentheses identifies the conditions under which the integrity constraints are checked and can be I-Insert, U-Update, or

D-Delete. If specified, the constraints are checked after all commands on the terminal screen have been performed.

Authcrization

QBE supports authorization constraints by filling in blank I.AUTH entries in the QBE dictionary. Authorization constraints can be printed (i.e. queried), updated, and deleted. The security in Figure 2.15 constraint above indicates that the specified user, USERID, may see fields F1 and F2. USERID may also update field F2 if and only if F2 < 25000. The value in the parentheses identifies the access capability and can be P-Print, I-Insert, D-Delete, or U-Update. The owner of a relation can confer access rights to others.

Concurrency and Locking

Concurrent reading of the database file is permitted, but only one user may have write access to the file at one time.

Physical Structure

No information about the underlying physical structure of the QBE database system was available in any of the literature surveyed.

SP	S#	SNAME	STATUS	CITY
P.SX P.SY			> 20	PARIS

QUERY: Get the supplier numbers for suppliers who are located in Paris or have STATUS > 20 (or both).

Figure 2.13 - Sample QBE Query

	F1	F2	F3	F4
I.CONSTR (I,U) I.CONSTR (D)	> 0		< F2	[LONDON,PARIS]

Figure 2.14 - Specification of QBE Integrity Constraint

	F1	F2
I.AUTH (U),USERID I.AUTH (P),USERID	EX	SZ < 25000

Figure 2.15 - Specification of QBE Security Constraint

Relational Database Management System (RDBMS)

Overview

The Relational Database Management System (RDBMS), described in [25], was developed by the British Broadcasting System, International Computers Limited (ICU), and Southhampton University to investigate the use of a wide variety of database techniques to produce large management information systems.

The RDBMS software consists of a number of autonomous processors linked by a communications network, as is shown in Figure 2.16. The Primary Processor hosts the system software that provides facilities for retrieval, manipulation, definition, and control of data.

The Demon Processors perform a variety of system support tasks, including processor communication and database restructuring.

Any number of Subsidiary Processors are used to execute user-defined functions requiring large quantities of software. These user functions allow a system designer to extend the system with new input and output formats, user-defined data and storage mappings, and additional integrity enforcement facilities.

Finally, one or more Language Pre-Processors permit an application program written in a high-level language to access and manipulate the RDBMS database.

User Interface

User access to RDBMS is through a family of languages known as the Control Sub-Languages (CSLs). The languages allow a user to access the system either directly via a terminal, or indirectly through an application program. In their simplest form, the languages permit a user to access all of the information within a database, along with the schema describing that information. Although the syntax of each CSL may vary, they all have similar mechanisms to specify integrity constraints, privacy restrictions, and define views.

RDBMS provides two main types of Control Sub-Languages. The Interactive Control Sub-Language permits a user to access the database via a simple query language. The Algorithmic Control Sub-Languages are extensions of standard computer languages that are intended for use in application programs. Each extension is designed to preserve and supplement the character of the host language.

RDBMS has three major levels of data descriptions: 1) entity sets, 2) encoded entity sets, and 3) stored entity sets. The first two are logical descriptions, while stored entity sets are physical descriptions of 1 and 2.

Logical Structure

Entity Sets

Each entity set is a tabular set of entries (called entities) taking the form of an n-ary relation of attribute/value pairs. Each attribute describes the role a particular domain will play in an entity set. No

attribute can have more than one value in an entity set. The logical associations between two or more entities in an entity set is through their attributes.

The definition of an entity set also includes a predicate. The predicate describes the contents of an entity set and formally defines the integrity checks to be performed on the entities in the set. Using the predicate, a system designer can force RDBMS to maintain 1) functional dependencies between two attributes in an entity set, 2) relationships between two or more entity sets, and 3) restrictions on domain values based on formal definitions of all domains.

When a system designer defines an entity set, he or she will recognize that there is one or more groups of attributes on which all other groups depend; such a group is called the Candidate Key of the entity set. When more than one candidate key exists for an entity set, one key must be designated as the Primary Candidate Key. The Candidate Keys are used to provide access to the entities in the set.

Encoded Entity Sets

Encoded entity sets restructure the entity sets based on estimates of the types and frequency of accesses made to groups of attributes, entities, and entity sets. The Information-Dependent Mapping Algorithms transform entity sets into encoded entity sets, while supporting the integrity predicates defined in the previous section. A single entity set may be mapped into one or more encoded entity sets using the relational operators, permutation, projection, join, restriction, selection, union, intersection, and difference, together with the non-relational operator order.

The primary structural difference between entity sets and encoded entity sets is that each encoded domain has a clearly-defined machine representation (e.g. character).

Physical Structure

Stored Entity Sets

Once the system designers have defined the contents and frequency of use of the encoded entity sets, they are now in a position to select indexes, access strategies, and physical storage structures. The results of this work are a number of stored entity sets and a collection of mapping algorithms. The designer can also insert any desired compression and conversion techniques.

Stored entity sets are transformed into one or more lower level stored entity sets using one of several Data-Dependent Mappings, based on capabilities of the underlying operating system. Examples of Data-Dependent Mappings supported by RDBMS are 1) serial, 2) keyed sequential, 3) indexed sequential, 4) random, and 5) in-core binary tree. The selection of a particular mapping will involve comparing efficiency measures for space usage, CPU utilization, and overall effectiveness.

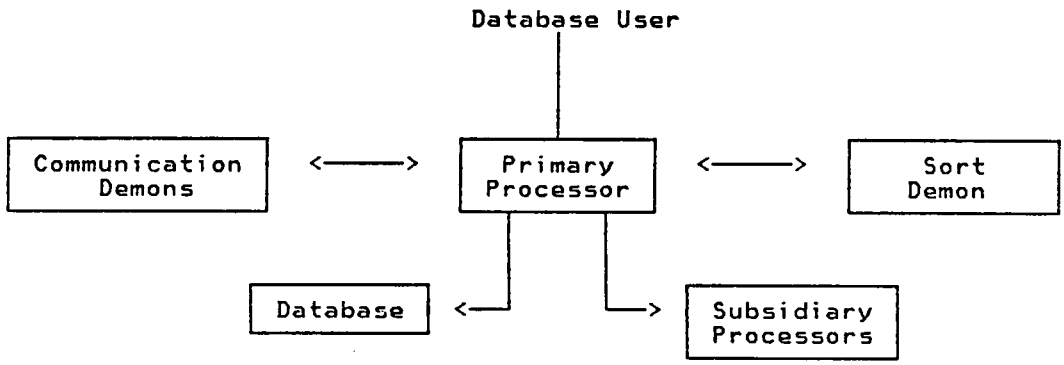


Figure 2.16 - Organization of RDBMS

System R

Overview

System R is an experimental relational database management system developed from 1975-1979 by the IBM Research Laboratory in San Jose, California to demonstrate the feasibility of a usable relational data management system that could respond to real database queries in a timeframe that was comparable to existing database systems. The goals of the experimental system, described in [1-4], [18], and [36], were to provide

1. a powerful, flexible data definition and query capability,
2. a versatile authorization and view subsystem,
3. a means of database recovery in the event of a hardware or software failure, and
4. a concurrent access support system that maintained database integrity.

System R was intended as a vehicle for research in database architecture; eventually, the System R prototype evolved into the SQL/Data System, an IBM relational database management product.

User Interface

A System R user views his or her database as a group of tables (base relations), static relations, and views (dynamically derived

relations). Like many other database systems, a relation is pictured as a row by column table, with no limit on the number of columns (attributes). Duplicate tuples are allowed in a relation.

User access to a System R database is through the SQL language (originally called SQUARE, then SEQUEL). With SQL commands, a user can define schema, query and alter the database, and impose integrity constraints. SQL commands may also be imbedded with PL/1 and COBOL programs.

Logical Structure

System R is implemented as a two-level machine. The top level, the Relational Data System (RDS), supports an external (user) interface which provides authorization, integrity enforcement, and support for alternate views of data. The RDS isolates the user from the underlying storage structures by providing a set of facilities for data retrieval, manipulation, definition, and control.

The bottom level, the Research Storage System (RSS), manages data access, space allocation, storage buffers, transaction recovery, system logging and recovery, and the automatic locking of data at varying levels of granularity. The RSS maintains indexes on selected fields of relations and links across relations. Figure 2.17 provides a pictorial overview (from [36]) of the components of System R.

The Relational Data System (RDS)

The RDS implements the Relation Data Interface (RDI), the principal external interface of System R. The RDI consists of a set of commands which provide high level data-independent facilities for the

definition, retrieval, and manipulation of relations and views. The RDI operators may be called from within a host programming language, or directly by an external dialog manager, the User Friendly Interface (UPI).

The RDI is primarily an extension of the SQL data sub-language. SQL can be used to insert, delete, and update single tuples or sets with a single command.

The RDS also contains an optimizer which, for a given set of SQL commands, tries to choose the optimum data access path from those provided by the Research Storage System.

Authorization Subsystem

The Authorization Subsystem allows users to enter the SQL commands GRANT and REVOKE to extend or deny access privileges to specific relations and views.

View Subsystem

The View Subsystem draws on the full query power of SQL to define alternate windows on the database (views). The View Subsystem also contributes to the overall security of System R since a view can be used to hide sensitive data from unauthorized users.

Integrity

Integrity constraints are defined by means of SQL assertions which may be specified at any time. Integrity is provided by an Assertion Feature which imposes constraints on tuples (e.g. CHARACTER(20)), that are enforced whenever tuples are inserted, deleted, or updated. The Assertion Feature also enforces "nulls not allowed" for designated fields, as well as uniqueness for designated field combinations. Two features of the SQL language not implemented in System R are assertions and triggers.

Physical Structure

The Research Storage System (RSS)

The RSS implements the Relational Storage Interface (RSI), the interface between the RDS optimizer and the underlying database storage structure. Using the set of RSI commands, the RDS has access to a variety of tuple operators so that the RDS optimizer can select the access path best suited to a particular SQL query. The RSS is also responsible for the physical storage and retrieval of relational data, the system recovery process, and the data locking subsystem which enables concurrent user access.

Physical Storage Management

All database information is stored in a collection of logical address spaces called segments which are used for storing user data, access path structures, internal catalog information, and intermediate results generated by the RDS. System R does its own storage management and I/O for segments by mapping logical segment spaces to physical extents on disk. Each segment consists of many equal-sized pages. Physical page slots on disk are allocated to segments dynamically upon first reference by checking and modifying the bit maps associated with the pages. Page slots are freed when access path structures are destroyed, or when the contents of a segment are destroyed. The RSS maintains a page map for each segment to identify all of the physical pages associated with a segment.

There are two main memory buffers. The first is used for segment page maps. The second buffer handles the actual data pages of a segment. Figure 2.18 shows how an individual tuple is accessed using its tuple identifier (TID). Note that at most two disk accesses are required to retrieve a specific tuple: the first loads a segment page map block, while the second loads the page containing the desired tuple.

The RSS provides three scanning modes: the relation scan, the image scan, and the link scan. The relation scan traverses the tuples in a relation as they are physically stored. The image scan accesses a relation by indexing a list of tuples. The link scan traverses pointers which connect a tuple to other related tuples. The decision to use a specific scanning mode is made by the optimizer and is communicated to the storage system using RSI operators.

In order to provide fast access to the tuples in a relation, all tuples of a relation reside within a single segment chosed by the RDS. This requirement clusters related tuples on a minimum number of pages and reduces the number of pages needed for a relation scan. A given segment, however, may contain several relations.

Associated with every tuple of a relation is a Tuple Identifier (TID). The TID uniquely identifes the tuple, as well as indicates the location of the tuple on disk. Each tuple is stored as a contiguous sequence of field values within a page. A prefix is stored with the tuple containing the relation identifier, the pointer fields (other TIDs) for link structures, the number of stored data fields, the number of pointer fields, and the length of each variable-length field. The TID is a concatenation of the page number within the segment and an a byte offset from the bottom of the page. At the offset slot is the byte location of the tuple in that page. This two-level scheme (TID to offset slot, offset slot to tuple), permits space to be compacted and tuples to be moved with only local changes to the offset slot pointers; offset slots are never moved from the bottom of each page so existing TIDs can still be used to access the tuples. An overflow scheme permits a tuple to be moved to another page should it become too large for its current page.

Recovery Subsystem

The goal of the Recovery Subsystem is to restore the database to a

consistent state in the event of a hardware or software failure. A consistent state is defined as a database that reflects only completed transactions. To aid in the recovery process, System R uses an image dump of the database, and a log of "before" and "after" changes. Should a transaction failure occur, the system simply processes the change log backwards, removing all changes made by the transaction; the change log is then corrected to contain only completed transactions.

Locking Subsystem

The Locking Subsystem provides a hierarchy of six granular lockable units, ranging from tuple locks to relational locks. Internal locks are also maintained on high traffic objects like the buffer pool and the change log.

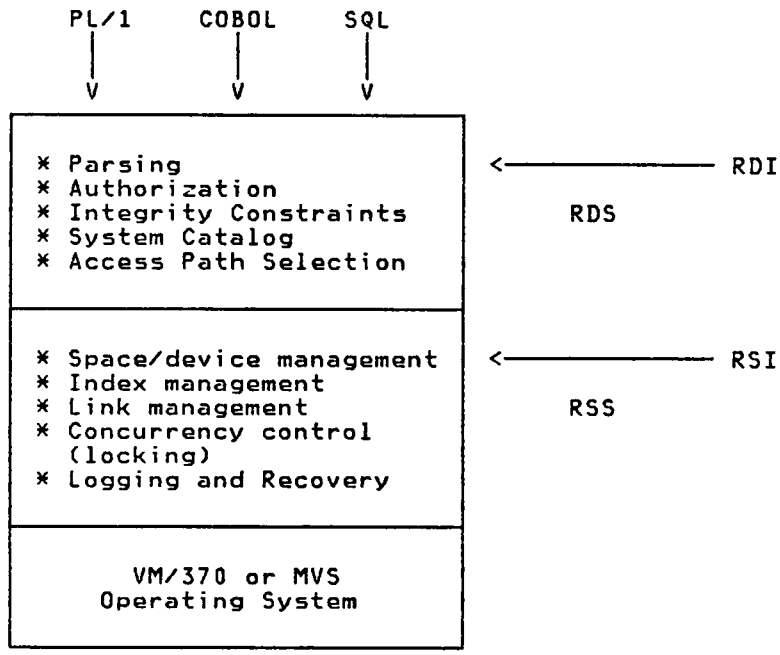


Figure 2.17 – Organization of System R

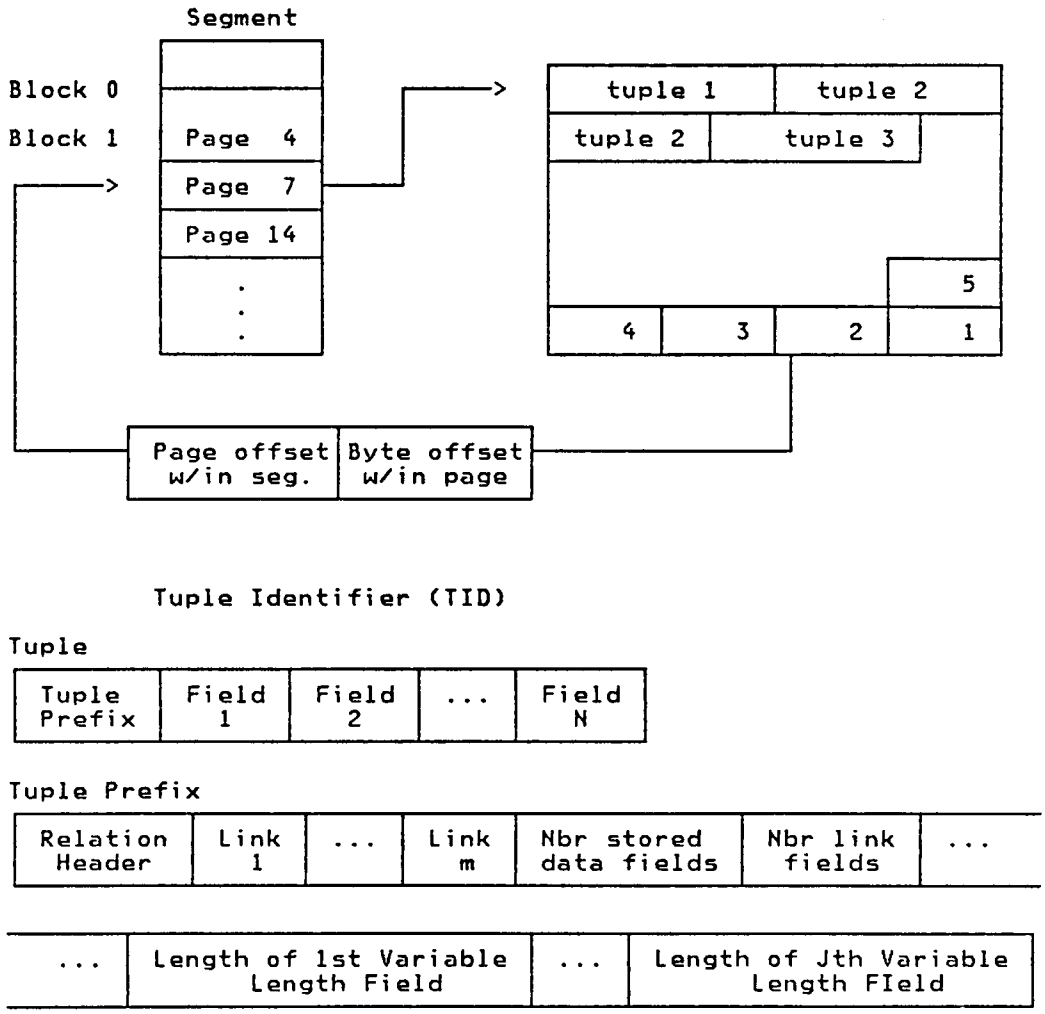


Figure 2.18 - System R - Accessing a Tuple Using its TID

ZETA

Overview

The ZETA relational database management system, described in [36], was developed as a prototype by the University of Toronto in Canada. The system has three distinct levels:

- Primitive Relational Level - implements elementary relational operations, inverted files, and the ability to "mark" subsets of relations,
- Intermediate Relational Level - implements derived relations and can combine elementary queries of the lower system into high-level relation operations, and
- The User Interface - uses a compiler-compiler for query language generation, a preprocesor-compiler for a host language system, and a semantic network for a natural language understanding system.

User Interface

The ZETA user has three means to access his or her database:

- DML is a Data manipulation language for program use within a PL/1 environment. The language syntax is a subset of SEQUEL. Commands are structured English commands within procedure calls, with a preprocessor providing the interface to the Intermediate Relational Level.

- The Query Language Interface is a compiler-compiler allows users to specify in a table the syntax of their language and to connect it to a set of modular semantic values. Subsets of the table will yield subsets of the language. This interface also has facilities for macro definition and expansion, and report generation.
- The Natural Language Interface interacts with user through TORUS, the Toronto Understanding System. A database user knows what is stored, not how. TORUS uses semantic networks as a basis for "understanding" dialog with a user, as well as knowing what information is stored in the database.

Logical Structure

The goal of the Intermediate Relational Level is to support primary and derived relations, and multi-relation queries. High-level optimization takes place when an operation or relation implementation can be performed in more than one way.

The Intermediate Relational Level is composed of three principal components:

- The Interpreter breaks down the input command data structure into a series of utility operations,
- A Set of intermediate-level schema procedures are invoked by the interpreter whenever data pertaining to derived or primary relations are to be stored or retrieved, and
- A Set of utility procedures to interface with the Primitive Relational Level.

Primary relations are those created by the Database Administrator. Derived relations are formed as a result of a join operation on multiple derived or primary relations, or from a subset of a single relation, formed by quantification. Two types of derived relations may exist: snapshots and automatic derivations.

A snapshot is a picture of a portion of the database at one time. Snapshots are implemented as marking relations. Prior to updating a tuple in the original relation, the old tuple is copied and logged into a separate section of the primary relation's file. The mark that pointed to the original tuple now points to the copy. The old tuple is then updated. Exceeding a threshold will cause the marking relation to be converted into an actual relation.

An automatic derived relation is a time-dependent relation. It is implemented as a derived relation formed as a marking relation. Changes to the original relation are logged by storing new tuples as part of an augmented primary relation, and marks pointing to the old tuples are made to point to the new tuples. Changes are time-stamped. The next time the derived relation is accessed, the definition is reexecuted on tuples not checked since the last access.

Physical Structure

The Primitive Relational Level provides a facility for manipulating relations. Actual (not derived) relations are stored on direct access files, with each tuple in the relation corresponding to a record in the file. System tables contain the names of relations, their domains, and other characteristics of each relation.

All commands at this lowest level operate on tuples within a single relation. Individual relations can be created, destroyed, locked, and unlocked. Individual tuples can be inserted, deleted, and updated. This level also queries schema tables which define the names and domains of relations.

Using a command at the Primitive Relational Level, relations may be "marked", a facility which is used to construct more complex operations (e.g. joins). A mark corresponds to a unary relation which stores indices of tuples of a relation which satisfy a Boolean qualification. Users may create or destroy marks, and retrieve tuples of a marked relation.

Summary

Having summarized the physical and logical structures of ten existing relational database systems, the following chapters will illustrate how the experimental disk-resident spatial information system developed for this project combines features of existing relational databases with new approaches to represent and manipulate information in a database.

CHAPTER 3 - PHYSICAL STRUCTURE OF THE DATABASE SYSTEM

This chapter describes the underlying physical structure of the disk-resident spatial information system. Major PASCAL data structures are identified, as well as means of representing the data types required by a sophisticated database system. Finally, a variety of VMS file structures are introduced that a database user or administrator can use to simplify the time required to bring up a database.

Note that the database system's physical structure is described first since this approach results in the fewest references to terms that will be defined in subsequent chapters.

Major Pascal Data Structures

Pages

A spatial database resides on disk as a single logically contiguous VMS file composed of equal-sized pages. Pages are read from and written to disk using modified copies of subroutines developed for the GIPSY Vision system [27]. Pages are read from disk into a fixed set of page buffers. Pages enter and leave the buffers under program control, based on a Least Recently Used (LRU) algorithm.

Each relation in the database is viewed as a collection of pages, linked by forward and backward pointers. The pointers permit efficient access

to pages with available space for new tuples, in addition to greatly simplifying the process of deleting a relation from the database.

Each page contains tuples for only one relation. This clustering results in fewer page accesses as subsequent tuples in a relation are accessed. Each tuple in a relation is identified by a Tuple Identifier (TID). The TID consists of a page number and a byte address of an offset slot for the tuple. The TID points to the tuple's offset slot and the offset slot contains the byte address of the tuple from the top of the page. This two tier system addressing scheme is similar to that used in System R [1-4]. By using offset slots instead of direct pointers to the tuples, a tuple can be relocated on a page without changing its TID. This feature will be especially useful if the database system is ever modified to permit variable-length tuples. The address in the offset slot is a byte offset to the tuple from the top of the page; the same is true of the offset slot pointer in the TID.

Figure 3.1 shows the format of a sample page. The offset slots are located at the bottom of each page. The number of offset slots per page is fixed for a relation and depends on the page size, the tuple size, and the number of bytes used for page overhead.

The next page and previous page fields are one word each and are used to delete a page when it is empty and to free all pages when a relation is deleted. As a relation requires additional pages, the database system initializes the page and adds it to the front of a doubly-linked list of pages for the relation. A field in the Relational Catalog entry for each relation points to the start of the linked list for a specific relation.

The next available and previous available page fields are also one word each and are used to maintain a linked list of pages having available space. These two fields are used when a new tuple is inserted into the

relation. If space is available on any already assigned page, that space will be used for the new tuple. Otherwise, space is taken from a new page. As tuples are deleted, the newly freed storage is placed on the list of pages with available space. A field in the Relational Catalog entry for each relation has the number of the first page with space available in the linked list for a specific relation.

The # bytes available on page field is a one-word field containing the total number of bytes that are unused on the page. This field was included in anticipation of eventually adding variable-length tuple support; the field is kept accurate, but is not used to determine whether a page has space available for a new tuple.

First available slot is a one-word field that contains a pointer to the first offset slot on this page representing unused space. Each offset slot for free space points to its successor, with the tail of the list flagged with a null pointer. The first available slot field is updated as tuples are added to or deleted from the relation. The next and previous available page fields point to pages with space available, while the first available slot pointer on each page points to the first offset slot representing available space.

The size of a page must be known when the interpreter is compiled. To have varying page sizes, separate copies of the interpreter must be used. "Suggestions for Future Work" on page 308 discusses the merits of allowing the page size to be dynamically specified by a database user or a system programmer.

Page Table

The page table is a bit map with one bit for each page that will be in the database file. If a bit in the map is zero, the corresponding database page is available for use. The number of pages occupied by the page table map depends on the maximum page table size in words. The page size is not involved in this computation.

Database pages are allocated sequentially, starting with the lowest numbered available page. Pages can be freed in any order. Should the interpreter encounter the situation where all database pages are allocated, the database file will be closed (as if DB_DOWN had been issued), and interpreter execution will be terminated.

The database system will have to be recompiled and the database rebuilt with a larger page size constant.

Page 1 of each spatial database file contains system variables. The page table map begins on page 2 of the VMS file containing the database. The number of pages the map occupies depends on its size and the page size. When a spatial database is up, the page table map resides in memory. When the database is closed, the page table is written to disk if any pages have been allocated or freed (i.e. the map has been modified). Two FORTRAN routines are used to manipulate the page table bit map since PASCAL's bit handling features are limited.

Page Buffers

The database system has a specified number of buffers, each of which can hold a database page. The buffers are allocated from the Workspace Area which is physically locked into the user's working set using VMS service

facilities. The goal of locking the page buffers is to avoid having VMS page out the database system's page buffers.

Associated with each buffer is an entry in the Buffer Reference Table which contains the following information:

1. The number of the page currently in the buffer,
2. a dirtybit indicating whether the buffer has been modified,
3. a next buffer pointer,
4. a previous buffer pointer, and
5. the byte offset to the start of the buffer within the Workspace Area.

Each entry in the Buffer Reference Table is a member of one of two linked lists. The first list ties together entries for buffers that are available (free). The second list links table entries so that the front of the list always points to the most recently referenced buffer; the tail of the list designates the buffer holding the next page to be written to disk (if necessary).

The number of page buffers must be specified when the interpreter program is compiled. "Suggestions for Future Work" on page 308 discusses the merits of allowing the database user or system programmer to specify this value.

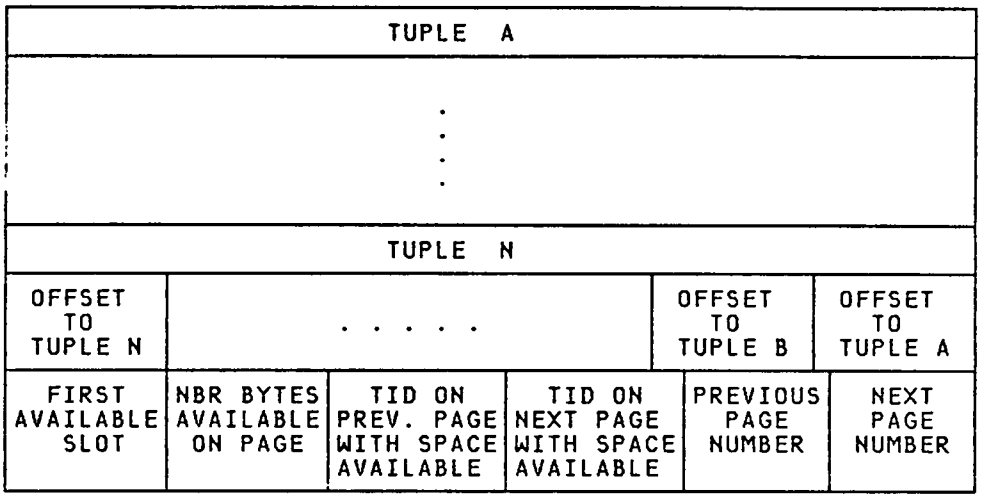


Figure 3.1 — Sample Database Page

Workspace Area

The Workspace Area is a large area of memory which is carved up as is required by database system modules. Most integers and all character strings (including vocabulary words) reside in the Workspace Area, as do the page buffers. The array representing the Workspace Area is locked in the user's working set in order to discourage VMS from including the workspace in its paging algorithms.

Storage is allocated from the Workspace Area in variable-length blocks on either a byte or a word boundary. The allocation procedure uses a modified version of the Boundary Tag method described in Horowitz and Sahni [23]. The allocation and deallocation algorithms were taken directly from that source, including the use of a head node and a circularly-linked free list.

Space is allocated in multiples of 4 bytes (one integer), even though the storage array is defined in bytes. There are no requirements that integers be allocated on word boundaries. This decision caused some implementation problems which are discussed in "Chapter 9 - Implementation Problems" on page 194. The minimum amount of space that can be requested by a database system module is one byte. The actual block allocated will be larger to account for the block overhead, plus padding the request to a word boundary.

The allocation procedure is based on a Next-Fit algorithm; the next free block large enough to satisfy an allocation request (plus the block overhead) will be allocated. The available Workspace Area is searched until either an adequate sized block is found, or the list of free blocks is exhausted. If a large free block must be split, the lower bytes are allocated. If the difference between the size of a free block and the required block size is minimal, the entire block will be

allocated. Subsequent searches for available blocks commence with the next free block following the last block examined (i.e. the search is circular).

Figure 3.2 provides a diagram of a free block of storage. All fields in the block are integers occupying one word each with the BDY and TAG fields sharing a word of storage. This means that 24 bytes of each block are used for overhead. For an allocated block, only the TAG, BDY, SIZE, and BTAG fields are used. Space for the LLINK, RLINK, and UPLINK fields must be reserved in an allocated block, however, so that those fields can be used when the block is later freed.

Active Relationlist

The Active_Relationlist (ARL) is a linked list in which each list entry contains all of the Relational Catalog and prototype information for a relation currently being accessed by the database system or by a database user. By referencing an ARL entry instead of the database file, time-consuming disk accesses are avoided. Up to 20 relations may have ARL entries at one time.

The ACCESSREL, ACCESSREL_FN, and ACCESSREL_SDS interpreter commands can be used to create an ARL entry for a relation. These commands

- Copy the information for the specified relation from the Relational Catalog to specific fields in the ARL entry.
- Create a memory-resident copy of the relation's prototype. The prototype defines the characteristics of each field in a tuple within the relation.
- Read the first tuple in the relation into a page buffer, and
- (Unordered lists only) Creates an ARL entry for the B-tree which will be used to determine the accessing order for tuples in the relation.

The RELEASEREL interpreter command frees an ARL entry for a relation by writing the Relational Catalog information back into the database if it has changed and if the relation was accessed in WRITEMODE; no writing to disk takes place if the relation was accessed in READMODE. For unordered relations, the ARL entry for the accessing B-tree is also released using RELEASEREL.

Only one tuple in a relation will be known to a given ARL entry at one time. This tuple is known as the "current tuple" and is the target of most ARL-related database commands. Database commands referencing an ARL entry must be cognizant of which tuple currently is in memory for a given relation. The NEXT_TUPLE database command can be used to access the tuple that is the successor to the "current tuple". When this successor is read into memory, it automatically becomes the new "current tuple". The ENDLIST? command can be used to determine whether the "current tuple" for a specific relation is the last tuple in the relation.

Note that there is no PREV_TUPLE command. When the database system was written, it was decided that such a function would not be needed. This function could easily be implemented, however, since each ARL entry contains a PREVIOUSTID field which has the TID of the predecessor to the "current tuple". Some type of FRONTLIST? command would also have to be implemented.

Database Data Types

The interpreter can support the types of data described below. All data types are represented by either an integer or a character string.

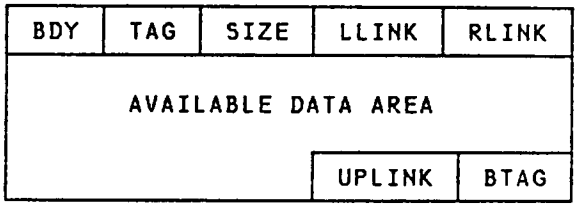
Real values are not currently permitted, although support could easily be added by treating a real variable as a character string that must be transformed each time the value is used in an arithmetic operation.

Boolean Fields

Boolean variables are treated as integers with a minimum value of 0 and a maximum value of 1. Since each boolean variable is an integer, 4 bytes of storage are used.

Character Fields

The interpreter allocates storage for character strings from the Workspace Area. Every string is preceded by a 4-byte length field. Strings can be of any length with the only restriction being the amount of storage available in the Workspace Area. All B-tree names, field names, prototype names, relation names and SDS names are implemented as character strings.



Field Descriptions:

BDY: Boundary Specification:
1 - Block was allocated on byte boundary
4 - Block was allocated on word boundary

TAG: Top Tag:
0 - Block is free
1 - Block has been allocated

SIZE: Number of bytes available in this block, including the overhead.

LLINK: Pointer to the block before this one in the free block list.

RLINK: Pointer to the block after this one in the free block list.

UPLINK: Pointer to the top of this block

BTAG: Bottom Tag:
0 - Block is free
1 - Block has been allocated

Figure 3.2 - Diagram of a Free Block of Storage

Integer Fields

Integer variables occupy 4 bytes of storage and can range from -2147483647 to +2147483647. These two limits are labeled -MAXINT and MAXINT by PASCAL. Note that although integers can be allocated from the Workspace Area, no length value precedes an integer field.

Field Names

The names of fields within a relation are implemented as character strings with a maximum length of 24.

Prototype Names

The names of all relational and SDS prototypes are implemented as character strings with a maximum length of 24.

Relation Names

Relation names are implemented as character strings with a maximum length of 24. The only distinction made between a relation name and any other type of character string occurs if the prototype has specified that a particular tuple field must contain a relation name. In that case, a check is made to determine whether the relation name string identifies an existing relation in the database.

SDS Names

SDS names are implemented as standard character strings. The only distinction made between an SDS name and any other type of character string occurs if the prototype has specified that a particular tuple field must contain an SDS name. In that case, a check is made to determine whether the SDS name string identifies an existing relation in the database. No check is made to ensure that the relation found is an SDS (versus some other type of relation). This decision was made for efficiency. To determine whether a given name identifies a relation requires searching the B-tree for the Relational Catalog. If the name was in the B-tree, the name is that of a relation. To determine the type

of the relation found, the Relational Catalog tuple for the relation would also have to be accessed and the relation type examined.

Command Names

Command names are implemented as character strings with a maximum length of 24.

File Names

File names are implemented as character strings with a maximum length of 24.

VMS File Structures

Input files

The interpreter can recognize six types of input files.

- A Spatial database file,
- An Interpreter command file,
- An Interpreter vocabulary file,
- A User vocabulary file,
- A Help file, and
- An Input command file.

Some files must exist, some are optional with pre-defined names, while others permit the user to assign the file name. More information about each type of input file is provided in subsequent sections.

Spatial Database Files

A spatial database resides on disk as a single logically contiguous VMS file composed of equal-sized pages that are read from and written to disk using modified copies of the GIPSY OSRDR and OSWTR subroutines (see [27] for details). The spatial database file contains a single spatial database composed of an arbitrary number of SDSs, relations, and prototype definitions. The DB_CREATE command creates a new spatial database file, while DB_UP opens a file containing an existing spatial database. Both of these commands require the user to specify the database file name, allowing multiple databases on a single VMS userid. In general, the spatial database files will not be printable since their logical record length will never be smaller than 256 bytes.

Interpreter Command File

The interpreter command file defines the commands that the interpreter can process. The file must be named 'RDSVOCB.DAT' and must exist on the userid of the interpreter user. If the interpreter command file cannot be found, interpreter execution will be terminated with a setup error.

The interpreter command file contains the following information about each permissible interpreter command:

- A command name that can be up to 24 characters long. All names are converted to uppercase and should not contain spaces.
- One of the following command precedence levels:
  - 0 denoting EXECUTEMODE (lowest precedence)
  - 1 denoting QUOTEMODE
  - 2 denoting LOOPMODE
  - 3 denoting COMPILEMODE (highest precedence)
  - * indicating that the corresponding case label number is used by an internal interpreter code and is not available for use.

  "Interpreter Precedence Levels" on page 103 describes the command precedence levels.

- The number assigned to the command that corresponds to an entry in a CASE statement used to distinguish between interpreter commands. A command number greater than 99 means that the command must be executed after a database (new or existing) has been brought up.

The command file name has been hardcoded in the VOCBINIT module and can be easily modified. In addition, it would be an easy modification to the database system to allow the database user to identify the command file to be used with his or her interpreter session. Using such an approach, a system administrator could enforce varying degrees of database security by assigning database users command files with various subsets of commands, based on the level of access required. In an academic environment, for example, a department office might be able to access and print information in the student database, but only the Registrar's office could create and delete student information. A user-supplied interpreter command file name would also permit users to tailor the interpreter command names through the user of synonyms (e.g. PRINTREL=PRINTR=PR).

The interpreter commands provided in the command file should not be confused with commands dynamically created by the database user through the use of the DEFINITION and END_DEF interpreter commands. Code to process commands in the interpreter command file must be present within interpreter modules.

"Appendix A - Sample Interpreter Command File" on page 319 has a copy of the interpreter command file used throughout this project.

Interpreter Vocabulary File

The Interpreter Vocabulary File, named INTRPVOCB.DAT, consists of interpreter commands that will be automatically processed when the interpreter is started. These sequences of interpreter commands may define constants, variables, and even new vocabulary commands. If the

interpreter vocabulary file is in a location accessible to all interpreter users, the database administrator can use it to define constants and vocabulary commands (e.g. new commands to perform relational operations like JOIN) that every database user will have access to. An interpreter vocabulary file does not have to exist in order to start the interpreter successfully.

"Appendix B - Sample Interpreter Vocabulary File" on page 321 has a sample interpreter vocabulary file. This sequence of interpreter commands defines the constants MAXINT and -MAXINT which represent the largest and smallest values a 4-byte integer field can take on in PASCAL.

User Vocabulary File

A User Vocabulary File allows a database user to define his or her own vocabulary commands that will automatically be processed when the interpreter is started. If a user vocabulary file is used, it must be named 'USERVOCB.DAT' and must be on the VMS userid of the database user. Note that each database user can define his or her own private vocabulary commands by including them in his or her USERVOCB.DAT file.

"Appendix C - Sample User Vocabulary File" on page 322 has a sample user vocabulary file that defines a new command, #OR. #OR performs the OR operation on a specified number of interpreter stack elements. Note that #OR is a user-defined command, while the OR interpreter command has been programmed into the interpreter.

Help Files

A Help File provides online Help information about a specific interpreter command's function, format, input parameters, and output values. An example of how to use the command is also included in the Help File. The HELP interpreter command can be used to display the Help File information for a specific command. Each Help file describes the

function of an interpreter command in terms of the input and output values.

All interpreter HELP files are named HELPXXX.HLP, where XXX is the CASE statement number assigned to the command in the interpreter vocabulary file. For example, the '+' command has been assigned CASE statement number 5; the Help file for '+', then, would be named HELP005.HLP. Help files can be added, deleted, or updated independent of the database system. Help files do not apply to vocabulary commands created from user or input vocabulary files.

"Appendix D - Sample HELP File" on page 324 contains the Help file for the '+' command which can be used to add two integers.

Input Command Files

An Input Command File is a sequence of interpreter commands that will be read and processed in response to an INPUT> interpreter command. The interpreter vocabulary file in Appendix B and the user vocabulary file in Appendix C could be input command files if the file names were changed.

Output Files

Using the >OUTPUT interpreter command, a user can direct the output from interpreter commands to either the terminal (normal mode of operation) or to a VMS file. Note that when output is directed to a file, the user will not know when the interpreter has finished processing a desired function since the prompt is also directed to the output file.

CHAPTER 4 - LOGICAL STRUCTURE OF THE DATABASE SYSTEM

This chapter describes the overall logical structure of the database system and identifies how the logical structure maps to the underlying physical structure described in Chapter 3.

Note that the database system to be described does not provide any of the following features:

- A sophisticated query language
- Query optimization
- Definition and manipulation of Views
- Concurrency
- System Recovery

Relations

The database system recognizes two types of user-defined relations: ordered lists and unordered lists. An ordered list requires tuples in the list to be accessed and maintained in the specific order they were added to the relation. An unordered list allows the database user to either specify an alternate ordering algorithm that will be used to access tuples in the list or to indicate that tuple ordering is unimportant. In all cases, a tuple ordering scheme exists, although it may be transparent to the database user.

In addition to ordered and unordered lists, a user may create relational prototypes to define what fields should be in a relation, SDS prototypes to define what relations should be in an SDS, and spatial data

structures, which are sets of logically related relations. A Relational Catalog will be created to store information about system and user-created relations. Finally, the database system will automatically create B-trees which can be used to indicate the sequence in which tuples in an unordered list should be accessed.

For simplicity, the database system treats B-trees, prototypes, spatial data structures, and the relational catalog as relations.

The relation name must be unique. The first two characters of a user-defined relation name cannot be '$$' because this prefix is only applicable for B-tree relations.

Ordered Lists

An ordered list (hereafter called an ordered relation) is a relation containing tuples that are accessed and maintained in the order that each tuple was inserted into the relation. To access tuple n+1, tuples 1 through n must first be read in; there are no exceptions to this rule.

Each tuple in an ordered relation has a field named NEXTTID which will be displayed when tuples in the relation are printed. The NEXTTID field identifies the TID of the next tuple in the relation. Since the tuples in an ordered relation must be accessed in a sequential manner, the database system can easily save the TID of the previous tuple, thus forming a doubly-linked list without requiring tuple space to define a backwards pointer.

The fields present in each tuple of an ordered relation are defined by the relation's prototype (including the NEXTTID field). Using the

relational prototype, the database can determine the length of tuples in the ordered relation, and the characteristics of every field in the relation.

Ordered relations are a convenient way to maintain tuples which are only meaningful if taken in a specific order. For example, the course of a river could be stored as an ordered list of tuples, with each tuple containing fields X and Y representing the longitude and latitude of the river at a specific location.

Unordered Lists

The tuples in an unordered list (hereafter called an unordered relation) can be accessed and updated in any order. Unlike ordered relations, unordered relation tuples have no NEXTTID field. Instead, the system uses a B-tree to determine which tuple to access next. The prototype for the relation identifies which fields in a tuple have corresponding B-trees; these fields, called indexed fields, may be used to designate the accessing order for tuples in the unordered relation. If no fields in the unordered relation have been identified as indexed, the database system will create and use a B-tree that orders the tuples in the relation based on the TID value of each tuple.

Tuples will be provided to a requester ordered by the value of a specific field in each tuple (e.g. order tuples by employee number). If no field is specified for ordering when the relation is accessed, the first indexed field will be used. If no field was designated as indexed when the unordered relation was created, tuples will be ordered by their TID value (which isn't necessarily chronological since the TID prefix is a page number).

Refer to "B-trees" on page 91 for more information concerning the relationship between unordered relations and B-trees.

Relational Catalog

The RELCATALOG relation is the Relational Catalog. Every ordered relation, unordered relation, spatial data structure, B-tree, SDS prototype, and relational prototype contains a tuple in this relation. RELCATALOG even contains a tuple for the RELCATALOG relation. The Relational Catalog tuple contains the following information for each relation:

- The relation name,
- The relation type (e.g. ordered),
- The number of the first page in the relation,
- The TID of an offset slot on the first page with space available,
- The number of keys (B-trees) or tuples in the relation,
-
  - Unordered Relations - The TID of the first tuple in the linked list of tuples in this relation,
  - Ordered Relations - The TID of the last tuple in the relation,
- The TID of the root node (B-trees) or first tuple in the relation, and
- The name of the prototype for this relation.

The Relational Catalog tuple for B-trees also has the following information:

- The maximum number of keys allowed in a B-tree node,
- An indication of whether the key is unique,

- The length of keys in the B-tree, and
- The field type of the key.

The Relational Catalog tuple for prototypes also has the following information:

- The length of tuples in the relations or SDSs that can be defined by this prototype, and
- The type of relation defined by the prototype.

RELCATALOG is maintained as an unordered relation. The B-tree relation, $$RELCATALOGRELNAME, is used to access individual RELCATALOG tuples using the RELNAME field of each tuple as the B-tree key. The relation type of RELCATALOG is RCATALOG to permit the system to maintain a minimal amount of database integrity by preventing the database user from modifying the RELCATALOG relation.

In order to optimize accessing the Relational Catalog information for database relations, the catalog information for the RELCATALOG relation is kept in memory as long as the database is up. An Active_Relationlist entry is used to contain the catalog information for RELCATALOG.

Relational Prototypes

A relational prototype describes the fields that should be present in a tuple when the database user creates a specific type of relation. Using the prototype, the database can determine the 1) type of relation to be created (ordered or unordered), 2) the length of tuples in the relation, 3) whether any B-trees should be created for fields in the relation, and 4) the characteristics of every field in the relation.

The advantage of the relational prototype approach is quickly realized if many relations of the same type have to be created (as in a geographical information system). The format of the relations only has to be described once - in the relational prototype. The prototype is also used to provide a certain amount of database integrity by ensuring that user-entered data does not violate the field definitions specified in the relation's prototype. Finally, the prototype is used to label each field in a tuple when a relation is printed.

The database system implements a relational prototype as an ordered relation, complete with a tuple in the Relational Catalog. When a relational prototype is created, the user identifies the type of relation defined by the prototype (ordered or unordered). This information is stored in the Relational Catalog tuple for the prototype. The initial tuple length of any relation defined by this prototype is zero. If the prototype defines an ordered relation, a field called NEXTTID will be automatically added to the prototype to identify the TID of the next tuple in the relation. As a database user adds a field to the prototype relation, the length of tuples in the relations that can be defined increases. For each field added to the prototype, a new tuple is added to the prototype relation. The order that fields are added to the prototype corresponds the order of the fields in a tuple.

The following fields will be in each tuple of a relational prototype:

- The field name,
- An indication of whether the field is indexed,
- The field length,
- The field type (e.g. character),
- If the field is indexed, an indication of whether the field value will be unique,
- For integers, the minimum and maximum values the field can take on, and

- The TID of the next tuple in the relational prototype.

Once a relation has been created using a particular prototype, adding a new field definition to the prototype will not cause the existing relation to be changed.

Relational prototypes are only meaningful for ordered and unordered relations created by the database user. Certain "special" relations have no prototype: relational prototypes, SDS prototypes, B-trees, and the Relational Catalog. The recursion involved to implement such a "meta-prototype" quickly proved to be prohibitive. Instead, internally-maintained prototypes are used. The internal prototypes are defined in such a manner that the same commands and subroutines that work for user-defined relations can generally be used for the "special" relations.

Whenever a relation is accessed, its relational prototype (whether internally or externally defined) is read into memory and is used to create a linked list of field definitions. The list is attached to the Active_Relationlist entry for the relation and will be used to minimize the time required to access or update a field in a tuple (i.e. no going to disk for field information).

SDS Prototypes

An SDS Prototype defines the names of relational prototypes describing relations which should comprise a specific type of spatial data structure. When an SDS is created using the prototype, the database system will automatically create each relation described by the SDS prototype and attach the relations to the SDS.

The database system implements an SDS prototype as a relation. An SDS prototype relation has a tuple in the Relational Catalog and an internally-maintained prototype. This means that the same interpreter commands and subroutines that work for user-defined relations can generally be used for an SDS prototype.

In order to avoid duplicate names for SDSs using the same prototype, the database system uses an internal naming convention to provide each new relation with a unique name. For example, suppose the STATE SDS prototype indicates that all spatial data structures of type STATE should consist of relations described by the STATE_AV, CITYLIST, and RIVERLIST relational prototypes. If SDS VIRGINIA is created using prototype STATE, relations VIRGINIASTATE_AV, VIRGINIACITYLIST, and VIRGINIARIVERLIST will also be created.

To avoid having to create an accessing B-tree as with unordered relations, SDS prototypes are ordered relations. Each tuple in the prototype contains a NEXTTID field identifying the the TID of the next tuple in the relation. There is also a field named RELNAME which contains the name of a relational prototype that will be used to create a relation for inclusion in an SDS of this particular type. An SDS prototype, then, is simply an ordered list of tuples, each containing NEXTTID and RELNAME fields.

Spatial Data Structures

A Spatial Data Structure (SDS) is a set of relations used to represent a geographical entity. The relations that are components of an SDS may be ordered relations, unordered relations, or even other spatial data structures. As defined, the concept of a spatial data structure is

flexible enough to represent both high-level geographic entities (e.g. states), and low-level entities (e.g. points). Since a spatial data structure is a relational structure, all of the primitive operations used in relational database systems are applicable to relations in a spatial data structure.

The database system implements a spatial data structure as an ordered relation. The SDS relation has a tuple in the Relational Catalog, and an internally maintained prototype. This means that the same interpreter commands that apply to user-defined relations can typically be used for an SDS.

Each tuple in an SDS has a field named RELNAME which contains the name of one relation that is a component of the spatial data structure. Since an SDS is an ordered relation, each tuple also has a NEXTTID field that identifies the TID of the next tuple in the SDS relation. The final field in an SDS tuple, CREATESW, is a boolean field indicating the disposition of the component relation when the SDS is deleted. If the CREATESW flag is 0, the component relation was automatically created when the CREATE_SDS interpreter command was issued. Since the relation has no meaning without the SDS, it will be deleted when a DELETE_SDS operation is performed for the SDS. A CREATESW value of 1 indicates that the component relation was created independently of the SDS and should not be deleted when the SDS is deleted.

B-trees

A B-tree relation is an M-way search tree that is used to provide rapid access to tuples in an unordered relation containing certain desired keys. A user can request that a B-tree relation be created for a

specific field in a relation by designating the field as INDEXED in the relational prototype. In that case, the value of the field is treated as a "key" in the B-tree and the TID of the tuple with the value as the data portion that corresponds to the key. Multiple fields in a relation may be INDEXED, although this will generally slow down the time required to update each tuple since all applicable B-trees must be updated. The B-tree key may be UNIQUE or NOTUNIQUE, again depending on the relational prototype definition. Note that key uniqueness only applies to one field in a tuple and cannot be used to ensure unique combinations of fields.

The B-tree for an indexed field will automatically be updated when 1) a new tuple is added to the relation, 2) an indexed field within a tuple is modified, and 3) a tuple is deleted from the relation.

The names of all B-tree relations start with '$$'; this is to make their relation names unique. The remainder of the B-tree name is a combination of the relation name and the name of the field in the relation that the B-tree is used to access.

The B-tree for an unordered relation can be accessed in two ways: 1) search the relation for tuples with specific field values, and 2) process each tuple in the relation one at a time, ordering the tuples by the value of a specific key. B-trees are well suited for the first approach since using a balanced B-tree is a fast means to search for specific keys. Using a B-tree for sequential processing is less than optimal, however, in a system with a limited number of page buffers since pages containing B-tree nodes must constantly be accessed and reaccessed as the search traverses each branch of the tree.

To speed up sequential access of unordered relations, a slight modification was made to the standard B-tree format. The TID of the data tuple associated directly with a specific B-tree key was replaced

by the TID of a link node tuple. This link node tuple, contains information about one tuple in the unordered relation that has the indicated key value. All of the link node tuples for the unordered relation are linked together, ordered by the key value. This means that to process each tuple sequentially in an unordered relation, the database system can use an ordered relation approach and follow the chain of link node tuples.

Duplicate keys are allowed in B-trees if NONUNIQUE was specified in the relational prototype description for a particular field. No new entries are made in the B-tree node for the duplicate key. Instead, the LASTFLAG field in the link node tuple for the first duplicate key will be set to 0, indicating that the next link node tuple is for the same key.

Figure 4.1 illustrates the format of B-tree and link node tuples, while Figure 4.2 shows how a simple B-tree with duplicate keys would be represented.

In order not to have to maintain 3 relations for each unordered relation (unordered relation, accessing B-tree, and link node tuple relation), the link node tuples for all relations are collected in one global relation named "BTREETIDLIST". This relation, described by the LINKNODEPROTO prototype, is not really a relation since it is composed of disjoint collections of tuples. It is considered a relation by the database, however, because all link node tuples are generated from the same set of pages. This was mainly a performance decision.

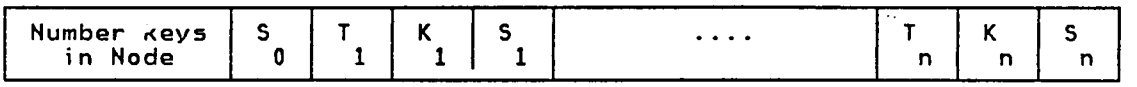
Tuples

All tuples added to a relation will be of fixed length, as defined by the relational prototype. Tuples in the relation may be explicitly ordered by the database user (e.g. any relation implemented as an ordered list), or the tuples may be implicitly ordered (e.g. unordered relations). The prototype indicates whether the relation will be ordered or unordered, and if unordered, which fields can be used to provide ordering. The prototype also designates the name and data type of each field in the tuple, in addition to identifying the permissible field values.

System Relations

The following four relations are automatically created when the DB_CREATE interpreter command is issued to create a spatial database. To maintain the integrity of the database, the user can only list or print these relations. The last 3 relations are described in subsequent sections. The RELCATALOG relation was described in "Relational Catalog" on page 86.

- RELCATALOG
- $$RELCATALOGRELNAME
- BTREETIDLIST
- LINKNODEPROTO



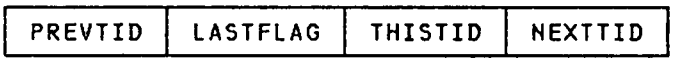
where:

S_0 = TID of the B-tree node contain keys less than K_1

T_n = TID of the Link node tuple containing TIDs of tuples with K_n

K_n = B-tree key

S_n = TID of the B-tree node contain keys between K_n and K_{n+1}



where:

PREVTID = TID of previous link node tuple for tuple in this relation. Will be 0 if the current tuple is first in the unordered relation.

LASTFLAG = 0-Next link node tuple represents tuple with same key
1-Next link node tuple represents tuple with different key

THISTID = TID of tuple in relation

NEXTTID = TID of next link node tuple for tuple in this relation.

Will be 0 if current tuple is last in unordered relation.

Figure 4.1 - Format of B-tree and Link Node Tuples

TID	Key
T9	5
T4	10
T6,T8	15
T1	20
T3	25

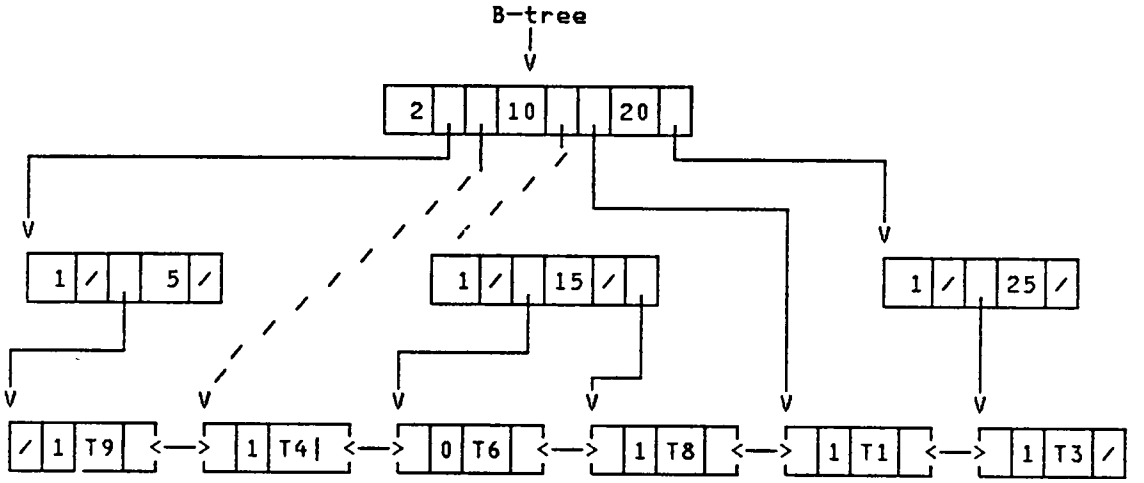


Figure 4.2 - Sample B-tree with One Duplicate Key

$$RELCATALOGRELNAME Relation

Since the RELCATALOG relation is an unordered relation, the $$RELCATALOGRELNAME B-tree will be used to access the TIDs of individual tuples in RELCATALOG. $$RELCATALOGRELNAME uses the relation name (RELNAME field of RELCATALOG) as the key in the B-tree. The data portion corresponding to the key is the TID of the tuple representing the relation within the RELCATALOG. All tuples added to the RELCATALOG relation will have their relation name and RELCATALOG TID added to the $$RELCATALOGRELNAME B-tree.

The TID of the RELCATALOG tuple for the $$RELCATALOGRELNAME relation will be written on page 1 of the database file since $$RELCATALOGRELNAME must be accessed before the TIDs of any other RELCATALOG tuples can be retrieved. The TID for $$RELCATALOGRELNAME will be read from the database file when the DB_UP command is used to bring up an existing database.

In order to optimize accessing the TIDs of tuples in the RELCATALOG, the Relational Catalog information for the $$RELCATALOGRELNAME relation is kept in memory as long as the database is up; no disk accesses will be required. An Active_Relationlist entry is used to contain the RELCATALOG information for $$RELCATALOGRELNAME.

BTREETIDLIST Relation

This relation contains link node tuples for ALL database tuples belonging to unordered relations. BTREETIDLIST is an ordered relation described by the LINKNODEPROTO prototype.

In order to optimize accessing and creating tuples for unordered relations, the Relational Catalog information for the BTREETIDLIST relation is kept in memory as long as the database is up. An Active_Relationlist entry is used to contain the catalog information for BTREETIDLIST. See "B-trees" on page 91 for more information about link node tuples and their relationship to B-trees.

LINKNODEPROTO Relation

LINKNODEPROTO is the prototype for the BTREETIDLIST relation. Using the NEXTTID and PREVTID fields, the link node tuples form a doubly-linked list containing the TIDs of all tuples in every unordered relation. Figure 4.1 provides a pictorial representation of a tuple in the LINKNODEPROTO relation.

System Variables and Tables

Variables

The following variables provide summary information about the spatial database system. The variable name is given in parentheses.

- # database pages currently in use (NBR_PAGES_INUSE)
- # tuples in database (NBR_TUPLES)
- # SDSs in database (NBR_SDS)
- # prototypes in database (NBR_PROTOS)

When a spatial database is not up, the variables reside on page 1 of the VMS file containing the database. When a spatial database is up, the variables reside in memory. The #TUPLES, #SDS, and #PROTOS interpreter commands can be used to obtain the values of the count variables. No system variable is used to record the number of relations in the database. This value is represented by the # KEYS/# TUPLES field in the RELCATALOG tuple for the $$RELCATALOGRELNAME relation.

The CURRENT_STATUS variable is used to record whether a previous low or high-level interpreter operation executed successfully. The DONEOK interpreter command can be used to query the value of CURRENT_STATUS.

The RELCBTTID variable contains TID of the RELCATALOG tuple for the $$RELCATALOGRELNAME relation. This value resides on page 1 of a

database file since the $$RELCATALOGRELNAME relation must be accessed before the TIDs of any other RELCATALOG tuple can be identified.

CHAPTER 5 - THE QUERY LANGUAGE INTERPRETER

This chapter describes the query language interpreter (first presented in [30]) and interpreter commands through which a user can interact with a spatial database.

Introduction

The query language interpreter provides the means through which a user can interact with a spatial database system. Using one or more commands in the query language, the user can perform any of the following function:

1. Access, create, delete, or modify individual tuples, relations, or spatial data structures in a selected database,
2. Perform arithmetic and boolean operations on stack entries,
3. Define variables, arrays, constants, and new commands to extend the query language,
4. Accept and direct interpreter input and output from an external VMS file,
5. Control the execution of the interpreter through the use of program control structures such as IF-THEN-ELSE, DO loops, and REPEAT-UNTIL loops, and
6. Perform a variety of debugging functions.

Differences

The following modifications were made to the query language interpreter documented in [30] and [49] in order to support the data types and operations needed by this project:

- All stack data types will be integers or pointed to by integer values,
- What was one vocabulary dictionary was split into two: an interpreter vocabulary and a user vocabulary. In the interpreter vocabulary, each command can be at most 24 characters long and have a entry indicating which CASE statement in the EXECUTE module performs the desired function. User vocabulary commands can be defined dynamically and can be of any length.
- Interpreter commands with CASE statement numbers greater than 99 cannot be executed until after a spatial database has been brought up. This greatly simplified error checking.
- Detailed error checking was provided for each interpreter command in order to ensure the integrity of the database and interpreter vocabulary.
- A common error subroutine was developed.
- The control structures described in [49] did not work in all cases. The overall control structure processing was rewritten to allow users to define sophisticated database commands to perform such functions as UNION and PROJECTION.

Interpreter Stack Types

An element on the interpreter stack will always be one of the following data types:

- ARL_ELEM - element is an index to a specific ARL entry,
- CHAR_ELEM - element points to a variable-length character string,
- INT_ELEM - element is an integer (includes booleans),
- IVOCB_ELEM - element is the index of a word in the interpreter vocabulary, and
- UVOCB_ELEM - element is the index of a word in the user vocabulary.

Note that the stack can only contain integers. For integer elements, the stack element is the integer. Otherwise, the stack element is an integer pointer value.

Interpreter Precedence Levels

The interpreter can be executing at one of four precedence levels:

- EXECUTEMODE - All input commands will be executed as they are entered by the database user.
- QUOTEMODE - A " was just entered and the next input value should be treated as a character string.
- LOOPMODE - A program control command (e.g IF) was just entered. Subsequent commands with a precedence level of COMPILEMODE will be executed immediately. Commands with a precedence of EXECUTEMODE or QUOTEMODE will be entered into the user vocabulary.
- COMPILEMODE - A command was just entered to either define or terminate the definition of a user vocabulary word.

EXECUTEMODE has the lowest precedence level, while COMPILEMODE represents the highest level. Whenever a command with a precedence level greater than the current interpreter level is encountered, the interpreter is switched to the higher level. The END_DEF, THEN, UNTIL, and +LOOP interpreter commands are the only commands that restore the interpreter precedence to its previous level.

The precedence level for a specific interpreter command is indicated in the interpreter vocabulary file.

Interpreter Commands

Commands in the interpreter vocabulary may be up to 24 characters in length. Most interpreter commands are long for two reasons. First, the name serves as a mneumonic for the interpreter function to be performed. For example, PRINTREL prints a relation. As indicated in the section entitled "Interpreter Command File" on page 79, shorter command names can be provided by modifying the interpreter command file. Second, an intelligent front-end program should eventually be added to the interpreter to provide a user-friendly means of accessing the database. The user interface, for example, might allow the user to access a spatial database through the use of sophisticated display menus. In such a case, the actual command names would be known by the front-end program, but not by the database user.

Interpreter commands can be free format. This means that indenting within control structures is permissible.

The database user may choose to use the DEFINITION and END_DEF interpreter commands to define user vocabulary commands. There is no restriction on the length of a user vocabulary command name.

Interpreter Command Descriptions

The succeeding sections describe each command in the query language that can be recognized by the interpreter. The commands are grouped into the following categories:

- Stack manipulation commands,
- Vocabulary manipulation commands,
- Program control commands,
- Database manipulation commands,
- Debugging commands, and
- Miscellaneous commands.

Commands within a category are arranged alphabetically. In all command descriptions, TOS represents the value on the top of the interpreter stack. TOS-1 (also called NOS) represents the stack element below TOS.

Command

Stack Manipulation Commands

The commands described in this section are general purpose primatives that manipulate the interpreter stack and perform simple arithmetic and boolean operations.

The commands listed below perform the same function as in [49], although the command error checking has been significantly improved. Refer to [49] for a detailed description of the command functions, inputs, and outputs.

+	Add the top two stack elements
-	Subtract the top two stack elements
*	Multiply the top two stack elements
/	Divide the top two stack elements
NOT	Complement the boolean value on top of the stack
ROT	Rotate the top 3 stack elements, clockwise
-ROT	Rotate the top 3 stack elements, counterclockwise
#ROT	Rotate the top x stack elements, clockwise
#-ROT	Rotate the top x stack elements, counterclockwise
#DROP	Drop the top x stack elements
#STACK	Retrieve # elements currently on stack
.STACK	Display contents of entire stack
.	Display top stack element
(STACK)	Replace TOS with stack(TOS)
CHAR?	Determine if TOS is character
INT?	Determine if TOS is integer
DUP	Duplicate the top stack element
DROP	Pop TOS from the interpreter stack
OVER	Push NOS onto the interpreter stack
SWAP	Swap positions of TOS and NOS on stack

The following interpreter commands, also described in [49], were modified to allow the relational operators to operate on two tuples within similarly defined relations. No other changes to the commands occurred. Refer to [49] for a detailed description of the command function, inputs, and outputs.

=	Determine if TOS = NOS
¬=	Determine if TOS ¬= NOS
>	Determine if NOS > TOS

<	Determine if NOS < TOS
>=	Determine if NOS >= TOS
<=	Determine if NOS <= TOS

Commands described on the following pages are all new.

Command: &

Function: AND the top two stack elements. The result replaces the input operands on the stack.

Stack Before:

Position	Description	Data Type
TOS	<integer 1>	INT_ELEM
TOS-1	<integer 2>	INT_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<integer 1> & <integer 2>	INT_ELEM
TOS-1	<stack element #1>	any data type

Example: Interpreter Command: &

```
Stack Before:        1                    <--- TOS
                     0
                     9

Stack After:         0                    <--- TOS
                     9
```

TOS was ANDed with TOS-1. The result, 0, replaced the two input operands.

Command: |

Function: OR the top two stack elements. The result replaces the input operands on the stack.

Stack Before:

Position	Description	Data Type
TOS	<integer 1>	INT_ELEM
TOS-1	<integer 2>	INT_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<integer 1> | <integer 2>	INT_ELEM
TOS-1	<stack element #1>	any data type

Example:

Interpreter Command: |

```
Stack Before:        1                    <--- TOS
                     0
                     9

Stack After:         1                    <--- TOS
                     9
```

TOS was ORed with TOS-1. The result, 1, replaced the two input operands.

Command: IDICT?

Function: Check the data type of the element on the top of the interpreter stack (TOS). If TOS is an interpreter vocabulary command, push a 1 (true) onto the stack. Otherwise, push a 0 (false) onto the stack. The element remains on the stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	0 - TOS is not an interpreter command 1 - TOS is an interpreter command	INT_ELEM
TOS-1	<stack element #1>	any data type

Example: Interpreter Command: IDICT?

Stack Before:	HELP 63	<--- TOS
Stack After:	1 63	<--- TOS

HELP is an interpeter command so a true value (1) is pushed onto the intepreter stack.

Command: UDICT?

Function: Check the data type of the element on the top of the interpreter stack (TOS). If TOS is a user-defined vocabulary element, push a 1 (true) onto the stack. Otherwise, push 0 (false) onto the stack. The element remains on the interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	0 - TOS isn't a user-defined vocabulary element 1 - TOS is a user-defined vocabulary element	INT_ELEM
TOS-1	<stack element #1>	any data type

Example:

Interpreter Command: UDICT?

Stack Before: 163 <--- TOS

Stack After: 0 <--- TOS
163

163 is not a user-defined vocabulary word so a false value (0) is pushed onto the interpreter stack.

Vocabulary Manipulation Commands

The primitives described in this section the interpreter vocabulary or access parts of the vocabulary.

The commands listed below perform the same function as in [49], although error checking was significantly improved. Refer to [49] for a detailed description of the command function, inputs, and outputs.

ALLOCATE
CONSTANT
FETCH
STORE
VARIABLE

Commands described on the following pages are all new.

Command: #IVOCB

Function: Push the current number of interpreter commands onto the interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# interpreter commands	INT_ELEM
TOS-1	<stack element #1>	any data type

Example:

Interpreter Command: #VOCB

Stack Before:	TEST1.DAT	<--- TOS
Stack After:	14	<--- TOS
	TEST1.DAT	

In the above example, 14 interpreter commands are defined.

Command: #UVOCB

Function: Push the current number of user-defined vocabulary elements onto the interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# user-defined vocabulary elements	INT_ELEM
TOS-1	<stack element #1>	any data type

Comments: The count of user-defined vocabulary elements will not reflect vocabulary words in the process of being defined (i.e END_DEF has not been encountered).

Example:

Interpreter Command:	#VOCB	
Stack Before:	TEST1.DAT	<--- TOS
Stack After:	32	<--- TOS
	TEST1.DAT	

In the above example, the user vocabulary dictionary has 32 elements. The UDUMP command could be used to determine what the 32 vocabulary elements are.

Command: DEFINITION

Function: DEFINITION causes a new user command to be created. DEFINITION changes the precedence of the interpreter to COMPILEMODE so that all subsequent input commands having a lower precedence will be compiled into the user vocabulary, instead of being executed immediately. The compilation continues until an END_DEF command is encountered. END_DEF lowers the interpreter precedence to its previous state.

DEFINITION commands may not be nested, but a definition can include already defined user vocabulary words.

Stack Before:

Position	Description	Data Type
TOS	Name of new command	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: "Appendix C - Sample User Vocabulary File" on page 322 has an example of how DEFINITION and END_DEF were used to define a user command.

Command: END_DEF

Function: Terminate the command definition started by the DEFINITION command. END_DEF has a precedence equal to COMPILEMODE so it will be executed, even when the interpreter is in COMPILEMODE. END_DEF restores the interpreter's precedence to its prevous level.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Comments: "Appendix C - Sample User Vocabulary File" on page 322 has an example of how DEFINITION and END_DEF were used to define a user command.

Command: FORGET

Function: Remove the word indicated by TOS from the user vocabulary.

Stack Before:

Position	Description	Data Type
TOS	Name of vocabulary word	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The FORGET command only applies to user-defined vocabulary elements. There is no command to delete commands from the the interpreter vocabulary.

Example: Interpreter Command: FORGET

Stack Before:	SIX	<--- TOS
	45	
Stack After:	45	<--- TOS

If the user vocabulary had the words FIVE, SIX, SEVEN, and EIGHT, issuing the FORGET command as indicated in this example would remove the word SIX from the vocabulary. The other three word would remain unchanged.

Command: IDUMP

Function: Dump all commands in the interpreter vocabulary.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example: Interpreter Command: IDUMP

Sample Output:

Interpreter Vocabulary Commands:

```
Command: CONSTANT     Case #:   1   State:      LOOPMODE
Command: VARIABLE     Case #:   2   State:      LOOPMODE
Command: FETCH        Case #:   3   State:   EXECUTEMODE
Command: STORE        Case #:   4   State:   EXECUTEMODE
Command: +            Case #:   5   State:   EXECUTEMODE
Command: -            Case #:   6   State:   EXECUTEMODE
Command: *            Case #:   7   State:   EXECUTEMODE
```

Command: UDUMP

Function: Dump all words in the user vocabulary.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example: Interpreter Command: UDUMP

Sample Output:

```
User-Defined Vocabulary Commands:

Index:   0   Command: SIX  State:  EXECUTEMODE
Index:   1         Entry:          6     Type: INT_ELEM
Index:   2         Entry:         15     Type: IVOCB_ELEM
```

Program Control Commands

The interpreter commands described in this section control the sequence of program execution. Although the commands were described in [49], they have been sufficiently modified and extended to warrant additional descriptions.

Program Control Structures

There are four program basic control structures: sequential, conditional (IF-ELSE-THEN), iterative loop (DO - +LOOP), and REPEAT-UNTIL loop. These structures control the manner in which interpreter commands will be executed.

Sequential Execution

Sequential execution is the normal mode of operation for the interpreter. In this mode, each command is executed as it is entered by the database user. This mode is the mode used when the interpreter is first started.

Conditional Execution (IF-ELSE-THEN)

Conditional execution allows the interpreter to execute or skip selected statements, based on a boolean value on top of the interpreter stack. The IF command identifies the start of the conditional control structure, while the THEN command terminates the control structure. The statements between the IF command and an ELSE command are commands that should be executed if the top of the interpreter stack has a true (1) value. Statements between the ELSE and THEN commands represent commands that should be executed if the top of the interpreter stack has a false (0) value. If no ELSE command is present, no false branch exists for the conditional test and a false stack value will cause control to be passed to the statement following the THEN command. IF-ELSE-THEN control structures may be nested up to 12 deep and may include any

interpreter commands, including those to create other program control structures.

Since the statements within an IF-ELSE-THEN structure cannot be executed until the entire structure is known, the statements must be stored until the THEN is encountered. To do this, the user vocabulary is used. When the IF command is encountered, the current interpreter state will be saved. If this IF-ELSE-THEN structure is the outermost program control structure (i.e. the previous interpreter precedence level was EXECUTEMODE), the interpreter is switched to LOOPMODE and a temporary command named IF00 is initiated in the user vocabulary. All subsequent input commands with a precedence level of EXECUTEMODE or QUOTEMODE will be inserted into the user vocabulary as part of the vocabulary word IF00. When the THEN command is processed, the saved interpreter state will be restored. If the previous interpreter state was EXECUTEMODE when the THEN command is processed, the IF00 vocabulary word will be executed. When an entry representing the outermost THEN command is read from the user vocabulary, a FORGET operation is performed to delete the IF00 vocabulary word.

Refer to "Conditional and Iterative Control Structure Example" on page 123 for an example of how the IF-ELSE-THEN control structure operates.

Execution of Iterative Loops (DO - +LOOP)

An iterative loop permits a sequence of interpreter commands to be executed a fixed number of times. The loop is delimited by the DO and +LOOP commands. DO initiates the control structure and provides the initial (TOS), final (TOS-1), and increment (TOS-2) values of the loop index variable. +LOOP terminates the loop. DO loops may be nested up to 4 deep and may include any interpreter commands, including those creating other program control structures.

Since the statements within a DO loop can be executed many times, the statements must be stored for the duration of the loop. To do this, the user vocabulary is used. When the DO command is encountered, the current interpreter state is saved and the interpreter is switched to LOOPMODE and a temporary command named DOOO is initiated in the user vocabulary. All subsequent input commands with a precedence level of EXECUTEMODE or QUOTEMODE are inserted into the user vocabulary as part of the vocabulary word DOOO. When the +LOOP command is processed, the saved interpreter state for the corresponding DO is restored. At that point, if the DO loop was the outermost program control structure (i.e. the previous interpreter level was EXECUTEMODE), the DOOO vocabulary word is executed, starting with the statement following the DO. Since the loop index variable is tested at the top of the loop, the DO loop may be executed zero times. When an entry representing the outermost +LOOP command is read from the user vocabulary, a FORGET operation is performed to delete the DOOO vocabulary word.

Since DO loops may be nested up to 4 deep, the four loop index variables may be obtained by the database user through use of the I, J, K, and L interpreter commands. I is the index variable for the outermost DO loop, while L is for the innermost loop. Executing any of these commands will cause the value of the specified loop index to be placed on the interpreter stack.

The BREAK command terminates execution of a loop by generating a branch to the statement following the +LOOP delimiter for the loop. The NEXT command causes control to be passed to the +LOOP statement for the currently active loop. Both BREAK and NEXT may be executed conditionally and can only be used to transfer control within the currently active loop.

Refer to "Conditional and Iterative Control Structure Example" on page 123 for an example of how the DO - +LOOP control structure operates.

Execution of Conditional Loops (REPEAT-UNTIL)

A REPEAT-UNTIL loop permits execution of a sequence of interpreter commands until a specified event has occurred. The loop is delimited by the REPEAT and UNTIL commands. REPEAT initiates the control structure, while UNTIL identifies the condition to test and terminates the loop. There are no limits to the number of REPEAT-UNTIL loops that can be nested.

As with conditional and iterative loop structures, the REPEAT-UNTIL control structure uses the user vocabulary to store and execute the statements with the loop.

REPEAT-UNTIL Example

Sample Input:	REPEAT DROP = UNTIL	
Stack Before:	2 6 B A A A 63	<--- TOS
Stack After:	63	<--- TOS

The REPEAT-UNTIL loop in this example deletes pairs of stack elements down to and including the case where the value of the TOS is the same as TOS-1. Note that the loop is executed once, even if the exiting condition is true on entry to the loop. This is, of course, one of the drawbacks of the REPEAT-UNTIL control structure.

Conditional and Iterative Control Structure Example

The example below of the DO - +LOOP and IF-ELSE-THEN control structures goes through the entire interpreter stack, performing the checks listed below:

- Entries whose depth in the stack is an even number are skipped.
- Entries whose depth in the stack is an odd number are printed.
- An entry whose value is 0 causes the loop to terminate immediately.

```
1
#STACK
1
DO
  I
  NOT
  0
  =
  IF
    NEXT
  ELSE
    DUP
    0
    =
    IF
      BREAK
    ELSE
      .
    THEN
  THEN
+LOOP
```

Refer to "Appendix B - Sample Interpreter Vocabulary File" on page 321 for a more sophisticated example of how the DO - +LOOP and IF-ELSE-THEN control structures can be used to create a new vocabulary word that can produce a copy of a relation.

Database Manipulation Commands

The interpreter commands described in this section allow a user to access and manipulate the information stored in a spatial database. Other than DB_CREATE and DB_UP, none of the database manipulation commands can be executed until a spatial database is available.

Table 5.1 attempts to correlate the functions of the database commands described in [49] with those defined by this system (described in much more detail later). The command from [49] is listed first. Note that due to the physical and logical organization of the database system, most of the database manipulation commands are different from those described in [49]. In some cases, such as with commands that operate with pointers and the interpreter stack, no equivalent command exists.

The commands listed in Table 5.2 are not in the memory-resident system: As with the commands in Table 5.1, each new command is described in more detail later.

Table 5.1 - Comparison of Interpreter Commands

ALLOC_RDS	Allocate and initialize an SDS header. Put a pointer to the header on the stack.
CREATE_SDS	Create a catalog entry for the SDS and use the SDS's prototype to create and attach to the SDS as many relations as were identified in the prototype.
ALLOC_REL	Allocate and initialize a relation header
CREATEREL	Create a catalog entry for a relation and, if the prototype indicates the relation is unordered, also create accessing B-trees for each indexed field in the relation.
DB_LOAD	Transfer an existing database or the definition for a new database (cannot tell difference) from disk to memory.
DB_UP	Read in the page table, system constants, and 4 system relations for an existing database. The remainder of the database will be accessed from disk when needed.
DB_UNLOAD	Transfer an existing database from memory to disk.
DB_DOWN	Write all modified database pages and the system constants to disk.
RDS_INDEX	Return the index to a SDS's entry in the RDS dictionary, given the name.
No equivalent	Users can retrieve information from the Relational Catalog, but not indexes to it.
REL_INDEX	Return the index to a relation's entry in the REL dictionary, given the name.
No equivalent	Users can retrieve information from the Relational Catalog, but not indexes to it.
(CATALOG)	Place a pointer to the header for an SDS on the interpreter stack.
No equivalent	Users can retrieve the name of a SDS's prototype only. The LISTREL command can display the Relational Catalog information.
(REL_CATALOG)	Place a pointer to the header for a relation on the interpreter stack.
No equivalent	Users can retrieve the name of a relation's prototype only. The LISTREL command can display the Relational Catalog information.
#CATALOG	Return the number of SDSs.
#SDS	Return the number of SDSs.

Table 5.1 - Comparison of Interpreter Commands (Continued)

#REL_CATALOG #RELS	Return the number of relations. Return the number of relations. This will include SDSs, prototypes, and accessing B-trees.
FIND No equivalent	Return a pointer to the header for SDSs and relations. Users can retrieve information from the Relational Catalog, but not indexes to it.
LIST_RDS LISTREL	List the information in an SDS's header List the information in an SDS's Relational Catalog entry.
LIST_REL LISTREL	List the information in a relation's header. List the information in a relation's Relational Catalog entry.
XLIST_RDS PRINTREL(SDS)	List the information in an SDS's header and print the name of each relation in the SDS. Print the name of each relation in the SDS. LISTREL can be used to get the Relational Catalog information.
XLIST_REL PRINTREL	List the information in a relation's header and print each tuple in the relation. Print the every tuple in a relation, labeling each field.
REL_ATTACH ATTACHREL	Attach a relation to an SDS (at the front). Attach a relation to an SDS (order unimportant).
NT_ATTACH BUILD_TUPLE	Add a tuple already defined on the interpreter stack to a relation pointed to by a relation header. Using the entries on the interpreter stack, add a tuple to a relation pointed to by an ARL entry. Validate that the tuple meets the integrity definitions for the relation, as defined by the relation's prototype. Add any indexed fields to their respective B-trees.

Table 5.1 - Comparison of Interpreter Commands (Continued)

INRELA?	Determine whether a tuple on the interpreter stack is in a specified relation.
No equivalent	Command could be done with ACCESSREL, relational operators, and RELEASEREL, but would be slow.
NAME	Given a pointer to the header for a relation, put the relation's name on the stack.
No equivalent	Users are expected to know the names of their relations. They do not have to be concerned about pointers and headers.
TYPE	Given a pointer to the header for a relation or SDS, return the type of the structure.
No equivalent	The PROTOTYPE can be used to return the name of the structure's prototype, but there is no way to determine what the structure is.
DIMEN	Return the dimension of an SDS or relation, given a pointer to the structure's header.
No equivalent	The ENDLIST? command can be used to determine when all tuples in a relation have been processed.
LENGTH	Return the length of an SDS or relation, given a pointer to the structure's header.
No equivalent	Users need to be concerned about prototype names, but will never directly update the length of tuples -- or have to worry about lengths since all of that information is stored in the prototype.
USE_CNT	Return the use count of an SDS or relation, given a pointer to the structure's header.
No equivalent	Relations, SDSs, and prototypes must be explicitly deleted. It is dangerous to automatically delete structures because there may be times when the user wants to retain the structure for later use (e.g. prototypes)
STRUCT	Return the structure pointed to by a header on the interpreter stack.
No equivalent	Users do not need to be concerned at this level.
LINK	Advance one node in a linked list
NEXT_TUPLE	Access the next tuple in any type of relation.
DATA	Return the data from the node pointed to by the top of the interpreter stack (specific field in the node).
VALUE_OF	Return the value of a specified field within a tuple (any field).

Table 5.1 - Comparison of Interpreter Commands (Continued)

RDS_DICT	Print the specified portion of the RDS dictionary.
PRINTREL	Print any relation, SDS, prototype, or B-tree in its entirity, including the Relational Catalog.
REL_DICT	Print the specified portion of the REL dictionary.
PRINTREL	Print any relation, SDS, prototype, or B-tree in its entirity, including the Relational Catalog.

Table 5.2 - Commands Not in Memory-Resident System

#PROTO	Determine the number of SDS and relational prototypes in the database.
#TUPLES	Determine the number of prototypes in the database.
ACCESSREL	Create an ARL entry for a relation. If the relation is unordered, also create an ARL entry for the appropriate accessing B-tree. Read in the relation's prototype.
ACCESSREL_FN	Create an ARL entry for an unordered relation and a specific accessing B-tree. Read in the relation's prototype.
ACCESSREL_SDS	Create an ARL entry for a relation that is attached to an SDS. Read in the relation's prototype.
ADDTO_RELPROTO	Add a field definition to a relational prototype
ADDTO_SDSPROTO	Add a field definition to an SDS prototype.
COPY_TUPLE	Copy a tuple from one relation to another, assuming the two relations have the same prototype.
CREATE_RELPROTO	Create a prototype for a relation. The prototype defines the fields that will be present in the relation, and identifies whether the relation will be ordered or unordered.
CREATE_SDSPROTO	Create a prototype for an SDS. The prototype defines the prototypes of relations that will be in the SDS.
DB_CREATE	Create a new database. The database will have the 4 system relations and a page table. The memory-resident system reads a database from disk and must rebuild the entire thing in memory; as such, it does not know whether the database it is building is new or old.
DELETE_PROTO	Delete a relational or SDS prototype.
DELETEREL	Delete a relation and, if unordered, its accessing B-tree. Note that the system in [49] required removing the relation's header from the linked list that was the REL dictionary, and deleting the tuples in the relation one by one.
DELETE_SDS	Delete an SDS and any relations that were not attached to the relation using the ATTACHREL interpreter command. See DELETEREL for an explanation of how [49] performed this function.
DELETE_TUPLE	Delete a tuple in a relation. If any of the fields in the tuple are indexed, delete the field values from their respective B-trees.

Table 5.2 - Commands Not in Memory-Resident System (Continued)

ENDLIST?	Determine whether the tuple just accessed was the last one in the relation. In [49], a zero pointer designates the end of a list.
PRINT_SDS	Print an SDS and all relations attached to the SDS.
PRINTREL_FN	Print all tuples in a relation, ordering the tuples by a specific indexed field.
PRINTREL_SDS	Print all tuples in a relation that is a component of an SDS. Using this command in place of PRINTREL avoids having to know the internal naming scheme for relations attached to an SDS.
PRINT_TUPLE	Print a tuple in a relation, labeling each field using the relation's prototype.
RELEASEREL	Release the ARL entry created by ACCESSREL, ACCESSREL_FN, or ACCESSREL_SDS. Update the Relational Catalog information, if needed.
REMOVEREL	Remove a relation that has been attached to an SDS. Note that in the memory-resident system, this can be done with pointers and linked lists.
SETVALUE	Set the value of a specific field in a tuple. If the tuple is indexed, update the accessing B-tree. Perform the integrity checks indicated by the relation's prototype. In the memory-resident system, this can be done using the interpreter stack and pointing to the specific field to be updated.

Command: #PROTOS

Function: Push the number of prototypes in the database onto interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# prototypes in the database	INT_ELEM
TOS-1	<stack element #1>	any data type

Comments: #PROTOS applies to both relational and SDS prototypes. The only way to determine how many of each specific type of prototype exists is to issue the STATS command.

Example:

Interpreter Command:	#PROTOS	
Stack Before:	VERMONT	<--- TOS
Stack After:	6	<--- TOS
	VERMONT	

This sequence indicates that 6 relational and SDS prototypes currently exist in the database.

Command: #RELS

Function: Push the number of relations in the database onto the interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# relations in database	INT_ELEM
TOS-1	<stack element #1>	any data type

Comments: Note that SDS prototypes, relational prototypes, and spatial data structures are considered relations and will be included in the count, along with ordered and unordered relations.

The count returned by #RELS will also include relations automatically created by the database system such as indexing B-trees and relations that are components of spatial data structures.

Example:

Interpreter Command:	#RELS	
Stack Before:	VERMONT	<--- TOS
Stack After:	22	<--- TOS
	VERMONT	

This sequence indicates that 22 relations currently exist in the spatial database.

Command: #SDS

Function: Push the number of spatial data structures in the database onto interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# SDSs in the database	INT_ELEM
TOS-1	<stack element #1>	any data type

Example:

Interpreter Command: #SDS

Stack Before:	VERMONT	<--- TOS
Stack After:	4	<--- TOS
	VERMONT	

This sequence indicates that 4 spatial data structures currently exist in the database.

Command: #TUPLES

Function: Push the number of database tuples onto interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	#tuples in database	INT_ELEM
TOS-1	<stack element #1>	any data type

Comments: This count includes tuples in all relations, SDSs, B-trees, and prototypes.

Example: Interpreter Command: #TUPLES

Stack Before:	VERMONT	<--- TOS
Stack After:	93	<--- TOS
	VERMONT	

This sequence indicates that 93 tuples currently exist in the database.

Command: ACCESSREL_FN

Function: Create an Active_Relationlist entry for an unordered relation. The entry will contain the Relational Catalog and prototype information for the relation.

Stack Before:

Position	Description	Data Type
TOS	Relation Name	CHAR_ELEM
TOS-1	Field Name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry created	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments: For simplicity, a vocabulary variable named after the relation will be created once ACCESSREL_FN has finished execution. The value of the variable is the number of the Active_Relationlist entry for the relation. The RELEASEREL command deletes the variable.

Example: Interpreter Command: ACCESSREL_FN

Stack Before:	STUDENTS NAME 14	<--- TOS
Stack After:	10 14	<--- TOS

The STUDENTS relation will be accessed using the NAME field to determine the order in which to access tuples in the relation. Using the GRADES field, if it existed, in place of NAME would allow a teacher to access the information in the STUDENT relation based on each student's grade.

Command: ACCESSREL_SDS

Function: Create an Active_Relationlist entry for a relation that is a component of a spatial data structure. The entry will contain the Relational Catalog and prototype information for the relation.

Stack Before:

Position	Description	Data Type
TOS	SDS name	CHAR_ELEM
TOS-1	Relation name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry created	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments: An internal database naming convention is used in order to avoid duplicate relation names in SDSs that use the same SDS prototype (e.g. the CITIES relation in the MARYLAND and VIRGINIA spatial data structures). The internal name is a concatenation of the SDS and relation names (e.g. MARYLANDCITIES). Relations created using this naming convention can be referenced directly, or though interpreter commands such as ACCESSREL_SDS and PRINTREL_SDS which will perform the name translation for the user.

For simplicity, a vocabulary variable named after the relation will be created once ACCESSREL_SDS has finished execution. The value of the variable is the number of the Active_Relationlist entry for the relation. The RELEASEREL command deletes the variable.

Example: Interpreter Command: ACCESSREL_SDS

Stack Before:	VIRGINIA	<--- TOS
	STATE_AV	
	15	
Stack After:	10	<--- TOS
	15	

This sequence creates an ARL entry for the STATE_AV relation that is a component of the VIRGINIA spatial data structure.

Command: ACCESSREL

Function: Create an Active_Relationlist entry containing the Relational Catalog and prototype information for the relation. For unordered relations, the B-tree for the first indexed field will be used to determine the order of tuples in the relation. If no tuple field is indexed, a B-tree was created when the unordered relation was created; this B-tree will be used by ACCESSREL. For ordered relations, tuples are accessed in the order they were inserted into the relation.

NEXT_TUPLE is automatically performed to read in the first tuple.

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry created	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments: For simplicity, a vocabulary variable named after the relation will be created once ACCESSREL has finished execution. The value of the variable is the number of the Active_Relationlist entry for the relation. The RELEASEREL command deletes the variable.

Example:

```
Interpreter Command:  ACCESSREL
Stack Before:         STUDENTS          <--- TOS
                      14
Stack After:          10                <--- TOS
                      14
```

Active_Relationlist entry number 10 has been assigned to hold information about the STUDENTS relation. A vocabulary variable named STUDENTS has also been created; its value is 10.

Command: ADDTO_RELPROTO

Function: Add a field definition to a relational prototype.

Stack Before:

Position	Description	Data Type
TOS	Prototype name	CHAR_ELEM
TOS-1	Field name	CHAR_ELEM
TOS-2	Field Type: I-Integer C-Character B-Boolean R-Relation name S-SDS name	CHAR_ELEM
When Field Type = 'B'		
TOS-3	Indexing Type: I- Field has a B-tree associated with it N- Field has no B-tree associated with it	CHAR_ELEM
TOS-4	Uniqueness Type: Field is not required if indexing type is not 'I'. U-Field in B-tree is unique N-Field in B-tree is not unique	CHAR_ELEM
When Field Type = 'I'		
TOS-3	Indexing Type: I- Field has a B-tree associated with it N- Field has no B-tree associated with it	CHAR_ELEM
TOS-4	Minimum Value	INT_ELEM
TOS-5	Maximum Value	INT_ELEM
TOS-6	Uniqueness Type: Field is not required if indexing type is not 'I'. U-Field in B-tree is unique N-Field in B-tree is not unique	CHAR_ELEM
When Field Type = 'C'		
TOS-3	Indexing Type: I- Field has a B-tree associated with it N- Field has no B-tree associated with it	CHAR_ELEM
TOS-4	Field Length	INT_ELEM

TOS-5	Uniqueness Type: Field is not required if indexing type is not 'I'. U-Field in B-tree is unique N-Field in B-tree is not unique	CHAR_ELEM

When Field Type = 'R' or 'S':

TOS-3	Indexing Type: I- Field has a B-tree associated with it N- Field has no B-tree associated with it	CHAR_ELEM
TOS-4	Field Length	INT_ELEM
TOS-5	Name of SDS or Relational Prototype	CHAR_ELEM
TOS-6	Uniqueness Type: Field is not required if indexing type is not 'I'. U-Field in B-tree is unique N-Field in B-tree is not unique	CHAR_ELEM

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: Once a relation has been created using a particular prototype, adding a new field definition to the prototype will not cause the existing relation to be changed. The merits of allowing such a dynamic reconfiguration of the database file are discussed in "Suggestions for Future Work" on page 308.

Field types SDS name ('S') and relation name ('R') are processed exactly like character data. Assigning unique field types allows the database system to perform some data checking when one of these fields are set by the user. Currently, the database system verifies that the relation or SDS named is in the

Relational Catalog, but does not distinguish between types 'R' and 'S'.

If a field has been designated as INDEXED, a B-tree will be created and maintained with the field value serving as the B-tree key.

Example: Interpreter Command: ADDTO_RELPROTO

Stack Before:	STATE_AV	<--- TOS
	POPULATION	
	I	
	N	
	1	
	100	
	43	
Stack After:	43	<--- TOS

The above example adds a field named POPULATION to the STATE_AV relational prototype. The POPULATION field is an integer ranging from 1 to 100. The values assigned to POPULATION should not be kept in a B-tree (i.e. not indexed).

Command: ADDTO_SDSPROTO

Function: Add the name of a relational prototype to an SDS prototype.

Stack Before:

Position	Description	Data Type
TOS	SDS prototype name	CHAR_ELEM
TOS-1	Relational prototype name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example: Interpreter Command: ADDTO_SDSPROTO

Stack Before:	STATE CITYLIST ABCD	<--- TOS
Stack After:	ABCD	<--- TOS

In the above example, relational prototype CITYLIST was added to the STATE SDS prototype. This means that all spatial data structures created with the STATE prototype will automatically have a relation with a prototype of CITYLIST created.

Command: ATTACHREL

Function: Attach a relation R to an spatial data structure S. Since SDSs are implemented as a relation, a tuple will be added to the SDS relation with the RELNAME field set the the relation name, R. The CREATESW field of the tuple will be 1, indicating that ATTACHREL was used to add the tuple to the SDS.

Stack Before:

Position	Description	Data Type
TOS	SDS name	CHAR_ELEM
TOS-1	Name of relation to be added to SDS	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example:

Interpreter Command: ATTACHREL

Stack Before:	TEMPREL VIRGINIA 69	<--- TOS
Stack After:	69	<--- TOS

In this sequence, the following tuple is created and added to the spatial data structure VIRGINIA:

- RELNAME = TEMPREL
- CREATESW = 1
- NEXTTID = TID of next tuple in SDS relation

Command: BUILD_TUPLE

Function: Create a new tuple and add it to a relation. The number of stack elements required for the command must be the same as the number of fields defined by the relation's prototype. The relational prototype will be used to check the integrity of the stack elements provided.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	.	
	1 element per field in the tuple	Appropriate data type
	.	
TOS-n	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments:

Example: Interpreter Command: BUILD_TUPLE

Stack Before:	10	<--- TOS
	86	
	95	
	221011234	
	Jane Doe	
Stack After:	10	<--- TOS

Assuming that Active_Relationlist entry 10 points to unordered relation STUDENTS with prototype STUDENTS_FORMAT, and that STUDENTS_FORMAT has the following fields:

- An indexed, unique character field called NAME,
- a non-indexed integer field called SSN,
- a non-indexed integer field called TEST1, and
- a non-indexed integer field called TEST2,

the command sequence in this example would create the following tuple:

- NAME = JANE DOE
- SSN = 221011234
- TEST1 = 95
- TEST2 = 86

Command: CLEAR

Function: Clear the following database performance measurement statistics:

1. # Prototypes in database
2. # SDSs in database
3. # Tuples in Database
4. # Relations in database
5. Page size (bytes)
6. Number page buffers
7. Number pages being used
8. # Reads from disk
9. # Writes to disk
10. # Database-recorded Page Faults
11. # System-recorded Page Faults
12. Amount of Buffered I/O
13. Amount of Direct I/O
14. Elapsed CPU time

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Comments: The STATS command can be used to print the current values of the performance measurement statistics. The SSTATS command prints an abridged version of the measurement statistics.

Command: COPY_TUPLE

Function: Copy a tuple from one relation to another. The relations must have the same prototype.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry for target relation	ARL_ELEM
TOS-1	# of ARL entry for source relation	ARL_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example:

Interpreter Command: COPY_TUPLE

Stack Before:	15	<--- TOS
	10	
	69	
Stack After:	69	<--- TOS

In this example, the tuple in the relation pointed to by ARL entry 10 was copied and added to the relation pointed to by ARL entry 15.

Command: CREATEREL

Function: Create a relation based on a relational prototype. The prototype defines the type of relation to be created, and describes the fields that will be present in the relation

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	Prototype name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The CREATEREL command can only create ordered and unordered relations. Other interpreter commands must be used to create other types of relations.

Example: Interpreter Command: CREATEREL

Stack Before: GRADES <--- TOS
GRADE_FORMAT
69

Stack After: 69 <--- TOS

Sample Output: Creating relation GRADES
Creating relation $$GRADES

The above example assumes that GRADES is an unordered relation requiring a B-tree ($$GRADES) to access individual tuples in the relation. GRADE_FORMAT is the prototype for the GRADES relation and will be used when the database needs to determine the characteristics of fields in the GRADES relation.

Command: CREATE_RELPROTO

Function: Create a prototype for a relation. The prototype defines the fields that will be present in a relation of this type.

Stack Before:

Position	Description	Data Type
TOS	Relational Prototype name	CHAR_ELEM
TOS-1	O - prototype for an ordered relation U - prototype for an unordered relation	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The CREATE_RELPROTO command can only create relational prototypes. Other interpreter commands must be used to create other types of database structures.

Example:

Interpreter Command: CREATE_RELPROTO

Stack Before:	STATE_AV	<--- TOS
	0	
	43	
Stack After:	43	<--- TOS

Sample Output: Creating relation STATE_AV

In this example, relational prototype STATE_AV was created. When the STATE_AV prototype is used to create relations, those relations will be ordered lists. Currently all tuples in relations defined by STATE_AV are 0 bytes long; the ADDTO_RELPROTO interpreter command can be used to increase the tuple length through the definition of new tuple fields.

Command: CREATE_SDS

Function: Create a spatial data structure based on an SDS prototype. The prototype contains the names of the relations that should be created and associated with this SDS.

Stack Before:

Position	Description	Data Type
TOS	Name of SDS	CHAR_ELEM
TOS-1	Name of SDS prototype	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The CREATE_SDS command can only create spatial data structures. Other interpreter commands must be used to create other types of database structures.
An internal database naming convention is used in order to avoid duplicate relation names in SDSs that use the same SDS prototype (e.g. the CITIES relation in the MARYLAND and VIRGINIA spatial data structures). The internal name is a concatenation of the SDS and relation names (e.g. MARYLANDCITIES). Relations created using this naming convention can be referenced directly, or though interpreter commands such as ACCESSREL_SDS and PRINTREL_SDS which will perform the name translation for the user.

Example: Interpreter Command: CREATE_SDS

Stack Before:	KANSAS STATE 88	<--- TOS
Stack After:	88	<--- TOS

Sample Output:

```
Creating relation KANSAS
Creating relation KANSASCITIES
Creating relation KANSASRIVERS
Creating relation $$KANSASRIVERS
```

The above example assumes that the STATE SDS prototype contains the names CITIES and RIVERS, and that RIVERS is an unordered relation requiring a B-tree to access individual tuples in the relation.

Command: CREATE_SDSPROTO

Function: Create a prototype for a spatial data structure. The prototype defines the prototypes of relations that will be in the SDS.

Stack Before:

Position	Description	Data Type
TOS	Prototype name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The CREATE_SDSPROTO command can only create SDS prototypes. Other interpreter commands must be used to create other types of relations.

Example: Interpreter Command: CREATE_SDSPROTO

Stack Before: STATE <--- TOS
69

Stack After: 69 <--- TOS

Sample Output: Creating relation STATE

In this example, SDS prototype STATE was created. STATE can be used to define spatial data structures. Right now, SDSs of type STATE will have no component relations.

Command: DB_CREATE

Function: Create a new spatial database, initializing all system relations, variables, and tables.

Stack Before:

Position	Description	Data Type
TOS	Database file name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The database file name must comply with the VMS file naming conventions.

Example:

Interpreter Command: DB_CREATE

Stack Before: DATABASE.DAT <--- TOS
36

Stack After: 36 <--- TOS

Sample Output:

```
Creating relation BTREETIDLIST
Creating relation $$RELCATALOGRELNAME
Creating relation RELCATALOG
Creating relation LINKNODEPROTO
```

In this example, a spatial database named DATABASE.DAT was created on the user's VMS id. Four system relations were initialized.

Command: DB_DOWN

Function: Bring down the database by performing the following functions:

1. Writing dirty pages still in memory to disk,
2. Releasing all Active_Relationlist entries currently in use, potentially causing the database to update the RELCATALOG information for the relations.
3. Write the performance measurement statistics to the current output file.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example:

```
Interpreter Command: DB_DOWN
Stack Before:        76          <--- TOS
Stack After:         76          <--- TOS
```

Command: DB_UP

Function: Bring up an existing spatial database. This process involves accessing all system relations, variables, and tables created by the DB_CREATE command.

Stack Before:

Position	Description	Data Type
TOS	Database file name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The database file name must comply with the VMS file naming conventions.

Example:

Interpreter Command: DB_UP

Stack Before: DATABASE.DAT <--- TOS
36

Stack After: 36 <--- TOS

Sample Output: Successful execution of DB_UP

Spatial database DATABASE.DAT was brought up. The four system relations were read in. The remainder of the database will be read in as needed.

Command: DELETE_PROTO

Function: Delete an SDS prototype or a relational prototype.

Stack Before:

Position	Description	Data Type
TOS	Prototype name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The DELETE_REL command can only delete SDS and relational prototypes. Other interpreter commands must be used to delete other types of relations.

Example: Interpreter Command: DELETE_PROTO

Stack Before: CITYLIST <--- TOS
69

Stack After: 69 <--- TOS

Sample Output: Deleting relation CITYLIST

The sequence above deleted the CITYLIST prototype. CITYLIST may have been an SDS prototype or a relational prototype.

Command: DELETEREL

Function: Delete a relation and all of the indexing B-trees used to access indexed fields in the relation.

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: The DELETEREL command can only delete ordered and unordered relations. Other interpreter commands must be used to delete other types of relations.

Example: Interpreter Command: DELETEREL

Stack Before: GRADES <--- TOS
69

Stack After: 69 <--- TOS

Sample Output: Deleting relation GRADES
Deleting relation $$GRADES

In the above example, GRADES was an unordered relation which used the $$GRADES B-tree to access individual tuples in GRADES.

Command: DELETE_SDS

Function: Delete a spatial data structure. All relations generated by the CREATE_SDS interpreter command will automatically be deleted. Any relations that were attached as a result of the ATTACH_REL command are not deleted (CREATESW field = 1).

Stack Before:

Position	Description	Data Type
TOS	Name of SDS	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example:

Interpreter Command: DELETE_SDS

Stack Before: VIRGINIA <--- TOS
22

Stack After: 22 <--- TOS

Sample Output:
```
Deleting relation VIRGINIA
Deleting relation VIRGINIACITY
Deleting relation VIRGINIARIVERS
```

In this example, no only was the SDS VIRGINIA deleted, but the CITY and RIVERS relations attached to VIRGINIA were also deleted.

Command: DELETE_TUPLE

Function: Delete the current tuple in the relation pointed to by a specific Active_Relationlist (ARL) entry.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1	any data type

Example:

Interpreter Command: DELETE_TUPLE

Stack Before:	-10 73	<--- TOS
Stack After:	-10 73	<--- TOS

In this example, the "current tuple" in the relation pointed to by active_relationlist entry 10 will be deleted. B-trees for indexed fields will also be updated.

Command: ENDLIST?

Function: Determine whether the current tuple is the last tuple in the relation pointed to by a specific Active_Relationlist (ARL) entry.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	0 - not last in relation 1 - last in relation	INT_ELEM
TOS-1	# of ARL entry	ARL_ELEM
TOS-2	<stack element #1>	any data type

Example: Interpreter Command: ENDLIST?

```
Stack Before:        10                  <--- TOS
                     14

Stack After:         0                   <--- TOS
                     10
                     14
```

In this example, the current tuple pointed to by Active_Relationlist entry 10 was not the last tuple in the relation. Knowing this information, the database user could issue a NEXT_TUPLE command to access the next tuple in the relation and make it the new "current tuple".

Command: LISTREL

Function: List all of the information stored in the RELCATALOG tuple for a specific relation.

Stack Before:

Position	Description	Data Type
TOS	Name of relation	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: Note that the term "relation" applies to B-trees, ordered relations, unordered relations, spatial data structures, SDS prototypes, and relational prototypes. Any type of relation can be listed using LISTREL.

Example: Interpreter Command: LISTREL

```
Stack Before:          RELCATALOG              <--- TOS
                       88

Stack After:           88                      <--- TOS

Sample Output:
Relation name:                  "RELCATALOG"
Relation type:                       RCATALOG
Front page of relation:                 3
First free page:                        0
Tuple Length:                         112
# Keys/# Tuples:                       4
RootTID/FirstTID:                    1025
Prototype name:                 "INTERNAL"

TID of indexing B-tree:               770
```

The Relational Catalog information for the RELCATALOG relation was listed.

Command: NEXT_TUPLE

Function: Access the next tuple in the relation pointed to by a specific Active_Relationlist (ARL) entry.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments: The ENDLIST? interpreter command can be used to determine whether the last tuple for the relation was just read in.

Example: Interpreter Command: NEXT_TUPLE

```
Stack Before:        10                <--- TOS
                     4

Stack After:         10                <--- TOS
                     4
```

In this example, the next tuple in the relation pointed to by Active_Relationlist entry 10 has been accessed. This tuple is the new "current tuple".

Command: PRINTREL

Function: Print all of the tuples in a relation. The relation's prototype will be used to determine the names, data types, and lengths of each field within the tuples.

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: Note that the term "relation" applies to B-trees, ordered relations, unordered relations, spatial data structures, SDS prototypes, and relational prototypes. Any type of relation can be printed using PRINTREL.

If PRINTREL is used with a spatial data structure, the output will consist of the names of the relations contained within the SDS. To print the contents of all relations contained within a specific SDS, PRINTSDS must be used.

Example:

Interpreter Command: PRINTREL

Stack Before:	GRADES 69	<--- TOS
Stack After:	69	<--- TOS
Sample Output:	See next two pages for examples	

*** RELATION: LINKNODEPROTO

```
FIELD_NAME = NEXTTID
INDEXING = N
FIELD_LENGTH =          4
FIELD_TYPE = I
UNIQUENESS = U
LOW VALUE =           0
HIGH VALUE = 2147483647
NEXTTID =       1538

FIELD_NAME = PREVTID
INDEXING = N
FIELD_LENGTH =          4
FIELD_TYPE = B
UNIQUENESS = U
LOW VALUE =           0
HIGH VALUE =          1
NEXTTID =       1539

FIELD_NAME = THISTID
INDEXING = N
FIELD_LENGTH =          4
FIELD_TYPE = I
UNIQUENESS = U
LOW VALUE =           0
HIGH VALUE = 2147483647
NEXTTID =       1540

FIELD_NAME = LASTFLAG
INDEXING = N
FIELD_LENGTH =          4
FIELD_TYPE = I
UNIQUENESS = U
LOW VALUE =           0
HIGH VALUE =          1
NEXTTID =         0
```

```
*** B-TREE: $$RELCATALOGRELNAME        ***
*** Keys are unique
*** Max key length:           28

Node:        1281    Depth in B-tree:           1

Number of keys is            4

Leftmost subtree is           0
Linktid sub  1 is        1025
TIDs of tuples with this key:
        770
Key sub       1 is $$RELCATALOGRELNAME
Subtree sub   1 is             0
Linktid sub   2 is        1026
TIDs of tuples with this key:
        769
Key sub       2 is BTREETIDLIST
Subtree sub   2 is             0
Linktid sub   3 is        1028
TIDs of tuples with this key:
        772
Key sub       3 is LINKNODEPROTO
Subtree sub   3 is             0
Linktid sub   4 is        1027
TIDs of tuples with this key:
        771
Key sub       4 is RELCATALOG
Subtree sub   4 is             0
```

Command: PRINTREL_FN

Function: Print an unordered relation using a specific field to determine the order that tuples will be printed. structure.

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	Field name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example:

Interpreter Command: PRINTREL_FN

Stack Before:	VIRGINIASTATE_AV MAJOR_CROP 69	<--- TOS
Stack After:	69	<--- TOS

The output for PRINTREL_FN will be similar to that provided earlier for PRINTREL.

Command: PRINTSDS

Function: Print a spatial data structure by performing a PRINTREL operation to print the contents of each relation in the SDS.

Stack Before:

Position	Description	Data Type
TOS	SDS name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: An internal database naming convention is used in order to avoid duplicate relation names in SDSs that use the same SDS prototype (e.g. the CITIES relation in the MARYLAND and VIRGINIA spatial data structures). The internal name is a concatenation of the SDS and relation names (e.g. MARYLANDCITIES). Relations created using this naming convention can be referenced directly, or though interpreter commands such as ACCESSREL_SDS and PRINTREL_SDS which will perform the name translation for the user.

Example: Interpreter Command: PRINTSDS

Stack Before:	VIRGINIA 69	<--- TOS
Stack After:	69	<--- TOS

The output for PRINTSDS will be similar to that provided for PRINTREL earlier.

Command: PRINTREL_SDS

Function: Print a relation that is a component of a spatial data structure.

Stack Before:

Position	Description	Data Type
TOS	SDS name	CHAR_ELEM
TOS-1	Relation name	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: An internal database naming convention is used in order to avoid duplicate relation names in SDSs that use the same SDS prototype (e.g. the CITIES relation in the MARYLAND and VIRGINIA spatial data structures). The internal name is a concatenation of the SDS and relation names (e.g. MARYLANDCITIES). Relations created using this naming convention can be referenced directly, or though interpreter commands such as ACCESSREL_SDS and PRINTREL_SDS which will perform the name translation for the user.

Example: Interpreter Command: PRINTREL_SDS

Stack Before:	VIRGINIA CITY 69	<--- TOS
Stack After:	69	<--- TOS

The output for PRINTREL_SDS will be similar to that provided for PRINTREL earlier.

Command: PRINT_TUPLE

Function: Print all of the fields in the current tuple pointed to by an Active_Relationlist (ARL) entry.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Comments: The term "relation" applies to unordered relations, ordered relations, SDS prototypes, relational prototypes, and SDSs.

Example: Interpreter Command: PRINT_TUPLE

Stack Before:
```
10                    <--- TOS
14
```

Stack After:
```
10                    <--- TOS
14
```

Sample Output:

```
FIELD_NAME = NEXTTID
INDEXING = N
FIELD_LENGTH =          4
FIELD_TYPE = I
UNIQUENESS = U
LOW VALUE =             0
HIGH VALUE = 2147483647
NEXTTID =        1538
```

All information in the current tuple identified by ARL entry 10 was printed.

Command: PROTOTYPE

Function: Replace the relation name at the top of the interpreter stack with the name of its prototype.

Stack Before:

Position	Description	Data Type
TOS	Relation name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	Prototype name	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Example:

Interpreter Command: PROTOTYPE

Stack Before:	VIRGINIA_AV 43	<--- TOS
Stack After:	STATE_AV 43	<--- TOS

In this example, the PROTOTYPE command has determined that STATE_AV is the name of the prototype for the VIRGINIA_AV relation.

Command: RELEASEREL

Function: Release the Active_Relationlist (ARL) entry for a relation. If any of the Relational Catalog information for this relation has changed (e.g. by DELETE_TUPLE), the RELCATALOG tuple for the relation will be updated.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	ARL_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: RELEASEREL can be used to free Active_Relationlist entries created by ACCESSREL, ACCESSREL_SDS, and ACCESSREL_FN.

Example:

Interpreter Command: RELEASEREL

Stack Before:	10 CITYLIST	<--- TOS
Stack After:	CITYLIST	<--- TOS

Command: REMOVEREL

Function: Remove a relation R that was attached to a spatial data structure S using the ATTACHREL command. Since an SDS is implemented as a relation, the tuple in S with RELNAME = R will be deleted. Relation R will also be deleted if the value of the CREATESW field in the SDS tuple is 0.

Stack Before:

Position	Description	Data Type
TOS	SDS name	CHAR_ELEM
TOS-1	Name of relation to be removed from SDS	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example: Interpreter Command: REMOVEREL

Stack Before:	TEMPREL VIRGINIA 69	<--- TOS
Stack After:	69	<--- TOS

Relation TEMPREL was removed from the VIRGINIA SDS. If TEMPREL was automatically created when VIRGINIA was created, TEMPREL will also be deleted.

Command: SETVALUE

Function: Set the value of a specified field in a tuple. The relational prototype will be used to verify the integrity of the new field value. Truncation will take place for character fields whose new value is too long. For indexed fields, the previous field value will first be deleted from the appropriate accessing B-tree before the new value is inserted in the B-tree.

Stack Before:

Position	Description	Data Type
TOS	New field value	any data type
TOS-1	Field Name	CHAR_ELEM
TOS-2	<ARL index>	ARL_ELEM
TOS-3	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS-2	<stack element #1>	any data type

Example: Interpreter Command: SETVALUE

Stack Before:	5400000 POPULATION 10 ABCD	<--- TOS
Stack After:	10 ABCD	<--- TOS

In this example, the value of the POPULATION field was set to 5400000 in the "current tuple" of the relation pointed to by ARL entry 10.

Command: STATS

Function: Print the database performance measurement statistics.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Comments: The CLEAR command can be used to reset the values of the performance measurement statistics.

Example: Interpreter Command: STATS

```
Stack Before:        NOW                  <--- TOS
                     HOPE
                     CITY

Stack After:         NOW                  <--- TOS
                     HOPE
                     CITY
```

Sample Output:

```
*********************************************

Todays date is       12-DECEMBER-1984
# Prototypes in database =          1
# SDSs in database =                0
# Tuples in Database =             13
# Relations in database =           4
Page size (bytes)    =           512
Number page buffers =             15
Number pages being used =          6
# Reads from disk=                 1
# Writes to disk=                  4
# Database-recorded Page Faults=   0
# System-recorded Page Faults=   500
Amount of Buffered I/O =          33
Amount of Direct I/O =             9

Elapsed CPU time =   1251 milliseconds

*********************************************

Successful execution of STATS
```

Command: VALUE_OF

Function: Place the value of a specific field in the current tuple for an Active_Relationlist (ARL) entry on the interpreter stack.

Stack Before:

Position	Description	Data Type
TOS	Field name	CHAR_ELEM
TOS-1	# of ARL entry	ARL_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	Value of field	any data type
TOS-1	# of ARL entry	ARL_ELEM
TOS-2	<stack element #1>	any data type

Example: Interpreter Command: VALUE_OF

```
Stack Before:        POPULATION          <--- TOS
                     10
                     CITY

Stack After:         53205               <--- TOS
                     10
                     CITY
```

In the above sequence, 53205 is the value of the POPULATION field of the current tuple for Active_Relationlist entry 10.

Debugging Commands

The commands described in this section are provided to facilitate the debugging of the interpreter system. As expected, most of the commands pertain to the database portion of the interpreter.

Command: PRINT#ARL

Function: Print a single entry on the Active_Relationlist (ARL). The entry contains a copy of the Relational Catalog and prototype information for a relation currently being accessed.

Stack Before:

Position	Description	Data Type
TOS	# of ARL entry	INT_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Example:

Interpreter Command: PRINT#ARL

Stack Before: 10 <--- TOS
CITY

Stack After: CITY <--- TOS

Sample Output:

```
Relation name:                  "BTREETIDLIST"
Relation type:                    ORDEREDLIST
Front page of relation:                     4
First free page:                            4
Tuple Length:                              16
# Keys/# Tuples:                            4
RootTID/FirstTID:                           0
Prototype name:                 "LINKNODEPROTO"
TID of last tuple:                          0
Access mode:                    WRITEMODE
Current TID:                                0
New tuple flag:                 FALSE
# ARL uses:                                 1
TID in RELCATALOG:                        769
Tuple offset:                           25354
Buffer number:                             15
ARL Use Type:                           ARL_INUSE
Next TID:                                   0
Previous TID:                               0
Hasindexed_fields:              FALSE
Prototype Definition:

No fields defined in the prototype
```

Command: PRINTARL

Function: Print the Active_Relationlist (ARL). There is one entry in this list for each relation currently being used by the spatial database system. Each entry contains a copy of the Relational Catalog and prototype information for a relation, in addition to some flags and counters specific to the relation.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example:

Interpreter Command:	PRINTARL	
Stack Before:	CITY 10	<--- TOS
Stack After:	CITY 10	<--- TOS
Sample Output:	Sample output on next page	

Relation name:	"LINKNODEPROTO"
Relation type:	RELPROTOTYPE
Front page of relation:	6
First free page:	6
Tuple Length:	55
# Keys/# Tuples:	4
RootTID/FirstTID:	1537
Prototype name:	"INTERNAL"
TID of last tuple:	1540
Length of tuples defined:	16
Type of relation defined:	ORDEREDLIST
Access mode:	WRITEMODE
Current TID:	1537
New tuple flag:	FALSE
# ARL uses:	1
TID in RELCATALOG:	772
Tuple offset:	26890
Buffer number:	12
ARL Use Type:	ARL_INUSE
Next TID:	1538
Previous TID:	0
Hasindexed_fields:	FALSE
Prototype Definition:	

Field name	Length	Low Value	High Value	Field Type	Unique?	Index
"FIELD_NAME"	24	0	0	C	YES	NO
"INDEXING"	5	0	0	C	NO	NO
"FIELD_LENGTH"	4	4	2147483647	I	NO	NO
"FIELD_TYPE"	5	0	0	C	NO	NO
"UNIQUENESS"	5	0	0	C	NO	NO
"LOW VALUE"	4	-2147483647	2147483647	I	NO	NO
"HIGH_VALUE"	4	-2147483647	2147483647	I	NO	NO
"NEXTTID"	4	-2147483647	2147483647	I	YES	NO

Command: PRINTBRT

Function: Print the Buffer Reference Table. This table contains information about all database pages that are currently in memory.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example: Interpreter Command: PRINTBRT

Sample Output:

```
** The following buffers are currently in use **
   Buffer: 15 Pageno:   3   Dirty:   TRUE  Status: INUSE  Offset: 25353
   Buffer: 12 Pageno:   6   Dirty:   TRUE  Status: INUSE  Offset: 26889
   Buffer: 14 Pageno:   4   Dirty:  FALSE  Status: INUSE  Offset: 25865
   Buffer: 13 Pageno:   5   Dirty:  FALSE  Status: INUSE  Offset: 26377

** The following buffers are currently free **
   Buffer:  1 Offset: 32521
   Buffer:  2 Offset: 32009
   Buffer:  3 Offset: 31497
   Buffer:  4 Offset: 30985
   Buffer:  5 Offset: 30473
   Buffer:  6 Offset: 29961
   Buffer:  7 Offset: 29449
   Buffer:  8 Offset: 28937
   Buffer:  9 Offset: 28425
   Buffer: 10 Offset: 27913

   Buffer: 11 Offset: 27401
```

Command: PRINTFREE

Function: Print a listing of free blocks within the Workspace Area.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Example: Interpreter Command: PRINTFREE

Sample Output:

Print of Work Space

* Free Blocks *

Block Addr	Left Link	Block Size	Tag	Right Link	Data Area	Uplink	Bottom Tag
1	25	24	1	25	17 to 27	1	1
25	1	33056	0	1	41 to 33083	25	0

Total available space = 33080 bytes

Miscellaneous Commands

This section describes the interpreter commands that do not fit in any of the other sections. These commands generally make it easier for a user to interact with the interpreter.

The commands listed below perform the same function as in [49], although the error checking for each command has been significantly improved. Refer to [49] for a detailed description of the command function, inputs, and outputs.

```
***EOF
INPUT>
>OUTPUT
"
```

Commands listed on the following pages are all new.

Command:	DONE
Function:	Terminate execution of the interpreter. If a database file is still open, DB_DOWN is performed first to bring the database down gracefully.
Stack Before:	The stack is not used by this command.
Stack After:	The stack is not used by this command.
Example:	Interpreter Command: DONE Sample Output: *** Interpreter execution terminated *** Execution of the interpreter is terminated.

Command: DONEOK

Function: Push a true value (1) onto the interpreter stack if the system status variable is "DONEOK"; otherwise, push a false value (0) onto the stack.

Stack Before:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	0 or 1	INT_ELEM
TOS-1	<stack element #1>	any data type

Comments: DONEOK returns the status of the last command executed. This means that no commands should be issued between the DONEOK and the command that would have changed the status variable.

Example:

```
Interpreter Command:  DONEOK
Stack Before:         65          <--- TOS
Stack After:          1           <--- TOS
                      65
```

In this example, the previous interpreter command executed successfully and a true (1) value was placed onto the interpreter stack.

Command: HELP

Function: Provide information about using a specific interpreter command.

Stack Before:

Position	Description	Data Type
TOS	Command name as string	CHAR_ELEM
TOS-1	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

Comments: All HELP files are named HELPXXX.HLP, where XXX is the command number supplied by the vocabulary file. Command names for HELP must be entered as character strings by preceding the command with a quote (").

Example:

```
Interpreter Command:  HELP
Stack Before:         +                          <--- TOS
                      14
Stack After:          14                         <--- TOS
Sample Output:        Accessing file HELP005.HLP
                      *** contents of HELP005.HLP ***
```

Refer "Appendix D - Sample HELP File" on page 324 for the contents of HELP005.HLP.

Command:	PRINTOFF
Function:	Cause suppression of the message 'Successful execution of ____' after each interpreter command is executed.
Stack Before:	The stack is not used by this command.
Stack After:	The stack is not used by this command.
Comments:	The PRINTON command causes the interpreter to resume printing the message.

Command:	PRINTON
Function:	Resume printing the message 'Successful execution of ____' after each interpreter command is executed.
Stack Before:	The stack is not used by this command.
Stack After:	The stack is not used by this command.
Comments:	The PRINTOFF command suppresses the message.

Command: SSTATS

Function: Print an abbreviated version of the performance measurement statistics printed by the STATS command. Each statistic is printed only if it is not zero.

Stack Before: The stack is not used by this command.

Stack After: The stack is not used by this command.

Comments: The CLEAR command resets the measurement statistics.

Example:

Interpreter Command: SSTATS

Stack Before: 55 <--- TOS

Stack After: 55 <--- TOS

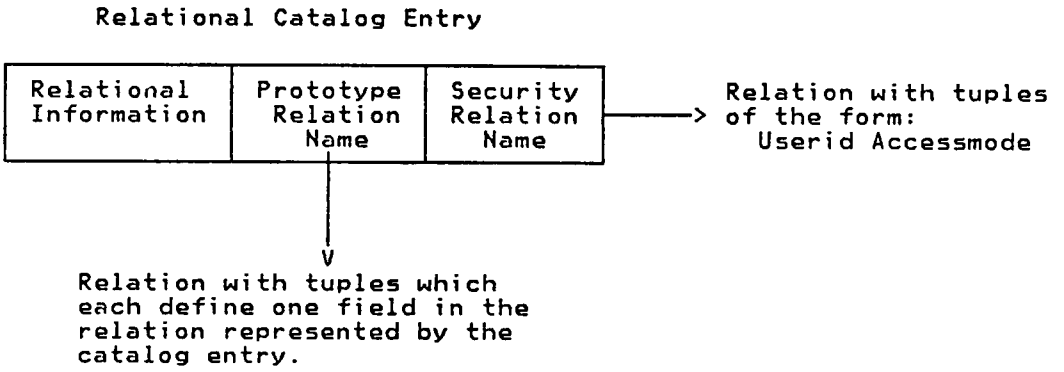
Sample Output:

```
Elapsed Time = 56 millisecond(s)
# System-related Page Faults = 3
# Database-recorded Page Faults = 0
# Reads from Disk = 1
# Writes to Disk = 1
```

CHAPTER 6 - DATABASE SECURITY, INTEGRITY, AND RECOVERY

Database Security

Currently the database system has no security protection features. Protection could be easily implemented, however, in a manner similar to relational prototypes.



When a user issues the ACCESSREL, ACCESSREL_FN, or ACCESSREL_SDS commands to access a relation, the security relation would be checked to ensure the database user has the correct authorization.

One problem with the suggested approach to database security is that a database administrator could not get a list of relations a specified user can access. Perhaps two lists could be maintained (relation by user and user by relation), but this has the potential to cause serious performance degradation since both lists must be kept up to date and consistent.

Database Integrity

The implementation of relational prototypes serves to not only simplify creating similar relations, but also to provide an efficient means of ensuring database integrity. Each relational prototype identifies the maximum length each field can be, the required field type, the maximum and minimum field values (for arithmetic values only), the uniqueness of a particular field (unordered relations only). Field uniqueness is validated for single fields only and cannot be used to guarantee the uniqueness of combinations of fields.

Since the database has no locking, it can only be used by one user at a time. There are no concurrency provisions.

Database Recovery

Currently, the database system has no recovery features. If an error is detected while a relation is being created or deleted, the current command fails, and no actions are taken to undo the portion of the command that already executed.

To prevent major recovery problems, however, two preventive measures were built into each database command:

- The input to each command is thoroughly checked before a command is executed. Although this is mainly an integrity measure, it did eliminate the requirement to back out once some integrity errors were encountered.
- Prior to performing the various operations required to execute a database command, the status of the previous operation is checked. This checking, although a nuisance in terms of programming and module readability, permits the database system to localize an

error, as well as prevent itself from propagating an error to subsequent operations.

A longer term solution to the recovery problem would be to build into the database system the ability to back out of a command once an error has been detected. Such an approach might entail creating an element that represents the inverse of the user's command and executing that inverse command should an error occur. Of course this approach does not work for destructive operations such as deleting relations.

CHAPTER 9 - IMPLEMENTATION PROBLEMS

GIPSY Modifications

The GIPSY subroutines (described in [27]) listed below had to be modified to have a "starting offset" parameter within the array being operated on. This modification was required because page buffers are allocated from a common Workspace Area, instead of having each buffer represented by an array, or substructure.

- OSRDR
- OSWTR
- MIFOLD

Buffer Use Conflicts

In the tests involving 10 page buffers, the problem occassionally arose when a page that had been accessed and was in a page buffer got paged out by a related operation such as searching a large B-tree. When it was recognized that this had happened, all copies of the interpreter were changed to reaccess the the desired page, causing it to be again read into the page buffer.

Bit Manipulation

Much of the information stored in relational prototypes could have easily been represented by bit variables (e.g. FIELD_INDEXED is a Yes/No decision). Unfortunately, PASCAL cannot perform bit manipulation. The end result was that many single bit fields were implemented as 4-byte integers, wasting valuable database disk space.

The only exception to the Bit=Integer rule is the page table map. References to the page table are fairly localized to two FORTRAN subroutines that allocate and free pages, addressing individual bits in the page table map.

Fixed-sized Pages, # Buffers

Since the arrays representing pages and page buffers had to be defined at compile time, the page size and number of page buffers could not be specified at execution time. This meant that in order to test various combinations of page sizes and # page buffers, separate copies of the database system programs had to be compiled for each combination. Every time an error was found in one copy, the other copies (which were identical except for the page size and # buffers constants), had to be recompiled.

"Suggestions for Future Work" on page 308 discusses the merits of providing a means to dynamically setting the two constants at execution time.

As with the page size and number of buffers, the expected number of pages in the database must be specified at compile time in order determine the size of the page table. If the number of expected pages is underestimated, the database program must be recompiled and the entire database must be recreated.

Representation of Data Types

When the physical structure of the database system was first planned, it was decided that integer fields in tuples would occupy only as many bytes as were needed to represent the field's maximum value. For example, if a relation's prototype stated that the maximum value for a specific field was 255, one byte would be allocated for the field within each tuple. Naturally bit fields would only occupy one bit.

This approach seemed ideal as a method of conserving disk space and reducing the database program's working set requirements. PASCAL, however, complicated the problem with its data type restrictions since it did not allow a storage location to be defined as both character and arithmetic. Unfortunately, this is just what was needed since the

meaning of tuple fields within a page buffer depended on how the relation's prototype defined each field.

The ideal solution provided by most programming languages that are used to write large system programs is to declare (or dynamically allocate) the page buffers as a character structure, and then declare bit, byte, halfword, and word structures whose definition could be based on specific offsets within a page buffer using some sort of pointer notation. If a field within a tuple was a two-byte integer, the halfword structure would be moved (rebased) to the correct location, and then used to reference the desired tuple field.

To get around PASCAL's data type limitations, the page buffers were implemented as character structures. Integers within page buffers were still represented as integers, but special utility routines had to be used to extract and insert integers that bypassed the type restrictions. The accessing routines, in part, could account for the slowness of some database commands since all integer fields in tuples required two subroutine calls to update (get/convert and put/convert). To avoid having separate extract/insert utility routine pairs to access bits, bytes, halfwords, and words, all integers within tuples, including boolean fields, were represented as 4-byte values.

With a concise means of representing integers and boolean values, the overhead used by the Boundary Tag method of storage allocation could be reduced from 24 to 8 bytes per unallocated block of storage. This improvement would greatly minimize storage usage, perhaps resulting in a significant improvement in system performance as VMS would have fewer database program pages in its working set.

CHAPTER 10 - PERFORMANCE MEASUREMENTS

This chapter presents the results of numerous tests to ascertain the performance of most of the database commands discussed in "Database Manipulation Commands" on page 125. Comparisons are made between the various relation types.

Introduction

Since this is a disk-resident database system, performance improvements will be identified by varying the page size and the number of page buffers recognized by the database system. The following combinations will be tested:

Page Size	# Buffers
256	10 20 30
512	10 20 30
1024	10 20 30

Note that since PASCAL must know the number of buffers and page size at compile time in order to generate addressing to the page buffers, nine separate copies of the database system code must be created and maintained each time an error is identified.

In the tests to be performed in this project, the working set size limit will be fixed at 150 and the working set quota at 300.

Approach to Testing

Commands To Test

To provide a comprehensive analysis of the database system, a large subset of database commands (described in "Database Manipulation Commands" on page 125) have to be tested for each page size/# buffers combination. For commands whose performance depends on the relation type (e.g. Ordered relations implemented as linked lists versus Unordered relations implemented as B-trees), all varieties of relation types have to be tested.

Average Performance

Since command performance also depends on whether the desired pages are already in a page buffer, executing each command only once would not be representative of how well the command performed. Instead, each command has to be executed enough times to identify the average system performance. It was determined (from observation) that executing each database command 50 times would provide a meaningful average measurement of system performance.

Startup Concerns

Other than the DB_UP and DB_CREATE interpreter commands, which in addition to the functions provided by the command, require VMS to access the majority of the database system modules (thereby inflating system page fault statistics), none of the other database commands were observed to have any startup peculiarities. Statistics from the first 5 commands executed were not significantly different from statistics for the last 5 commands executed.

Method of Operation

Since entering each database command and recording the performance

statistics manually would be both highly monotonous and error-prone, several shortcuts were taken. First, the DO and +LOOP interpreter commands were used to define an iterative loop which would 1) prepare for command execution, 2) clear the measurement statistics (CLEAR command), 3) execute the desired database command, and 4) print the values of the measurement statistics (SSTATS command). The loop would execute 50 times, satisfying the requirement to produce statistics which represented average performances. The DEFINITION and END_DEF interpreter commands were then used to define a user vocabulary word that would 1) issue the DB_UP command to access a specific spatial database, 2) set up whatever was necessary to perform the test, 3) execute the DO loop structure, and 4) issue the DB_DOWN command to save the database for a subsequent test. The entire vocabulary word definition was then put into a user vocabulary file, providing a test scenario for all 9 page size/# buffer combinations that would be run for each test.

The second and third shortcuts were necessitated by the inability to logon to the SDA lab system towards the end of each quarter. By submitting the test files to be run to the batch processor and directing the output (the measurement statistics) to an output file, testing could take place without incurring tremendous long distance phone charges (to dial into the SDS lab system from Raleigh, N.C.).

The final shortcut resulted in ordering the tests to be performed in such a manner that each test relied on its predecessors to provide the required operating environment. For example, the test to create 50 relational prototypes for ordered relations was run before the test to create 50 ordered relations. By ordering tests, the setup requirements for each test were minimized.

Statistics Recorded

The following statistics will be recorded each time a database command is executed during the tests:

- Amount of CPU time (in milliseconds) required to execute the command,
- The number of system-recorded page faults that occurred while the command was being executed,
- The number of database-recorded page faults that occurred while the command was being executed,
- The number of times the database system had to write a dirty page to disk, and
- The number of times the database system had to read a page from disk that was not already in memory.

At the conclusion of each test, the following statistics will be recorded to determine the effect the commands had on the database characteristics:

- Changes to the number of VMS blocks occupied by the database,
- Changes to the number of pages occupied by the database.
- Changes to the number of tuples in the database,
- Changes to the number of relations in the database,
- Changes to the number of prototypes in the database, and
- Changes to the number of spatial data structures in the database.

All of these values represent changes due to the particular command being tested and do not include differences incurred in setting up a test.

NOTE: The number of database page faults, reads, and writes will be rounded to the nearest tenths. As such, a value of 0.0 should not be construed to indicate that no instances of a particular action occurred (e.g. no database reads); instead, 0.0 should be interpreted as meaning that the action occurred fewer than 2.5 occurrences per 50 commands processed.

Approach to Analysis

Once all 9 combinations have been run for a particular test, the results will be analyzed to determine whether there were any significant differences between the various page size/# buffers combinations. A 'L' or 'H' will be placed next to the combinations that has the best and worst average execution times, since execution time directly measures how a user views database system performance. The other measurements will be analyzed to determine how they contributed to the average execution time. "Summary-Highest/Lowest Execution Times" on page 294 presents a summary of which combinations tended to receive the best and worst execution times.

The changes to database characteristics will also be analyzed. The key measures are # VMS blocks, # pages used, and # tuples used. The other measures should remain constant across all combinations and are provided for information only. In this category, the best combination is the one that uses the least disk space.

Contrasts will be made between tests involving the same database command, but different relation types (e.g. NEXT_TUPLE for ordered and unordered relations). "Summary-Ordered Relation vs Unordered Relation" on page 295 summarizes the differences.

Note that the performance of most database commands should significantly improve if successive commands reference the same relation (e.g. successively defining all fields in a relational prototype).

Test 1

Creating a New Database

Test: Determine the average amount of time required to initialize a new spatial database using the DB_CREATE command. Recall that a newly-initialized database has 4 system relations: RELCATALOG (unordered relation), $$RELCATALOGRELNAME (B-tree), BTREETIDLIST (ordered relation), and LINKNODEPROTO (relational prototype). A database page table will also be created on page 1 of the database file. Note that the page buffers are re-initialized with each DB_CREATE command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	1815.12	0.0	0.0	0.0	0.0
	256	20	1818.28	0.0	0.0	0.0	0.0
		30	1816.40	0.1	0.0	0.0	0.0
L		10	1177.04	0.0	0.0	0.0	0.0
	512	20	1818.79	0.0	0.0	0.0	0.0
		30	1819.84	0.0	0.0	0.0	0.0
		10	1816.34	0.0	0.0	0.0	0.0
	1024	20	1816.35	0.3	0.0	0.0	0.0
H		30	1859.58	574.5	0.0	0.0	0.0

Database Characteristics after Test 1

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	7	8	15
# Pages used	11	7	7
# Tuples	13	13	13
# Relations	4	4	4
# Prototypes	1	1	1
# SDSs	0	0	0

Analysis: The 512/10 combination had the best execution time. This may be because 512 bytes is the page size used by VMS, although one would expect this improvement to apply to the other 512-byte combinations. The fact that it produced the largest database program probably caused the 1024/30 combination to have significantly more system page faults. The other combinations tested appear fairly even in terms of execution time.

Since few pages are used by a new database, 10 page buffers proved adequate. All writing of the new pages is done when the DB_DOWN command is issued to save the new database.

Most of the pages required at startup are for entries in the RELCATALOG and $$RELCATALOGRELNAME relations. When a page size of 256 is used, few RELCATALOG tuples fit on each page, so more pages are needed than with larger page sizes. Since a fixed number of pages are needed for system information like the page table and system constants, however, larger (likely empty) system pages may cause more VMS blocks to be used for the 1024-byte combinations than their 512-byte counterparts.

Note that there is a direct correlation between the number of pages used, the block size, and the number of blocks used. For example, seven 1024-byte pages breaks down to slightly more than twice as many VMS 512-byte blocks.

Test 2

Closing a New Database

Test: Determine the average amount of time required to write a newly-initialized spatial database to disk using the DB_DOWN command. This process involves writing all 4 system relations created by DB_CREATE, the page table, and the system constants.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	8.05	0.0	0.0	0.0	11.0
	256	20	8.28	0.0	0.0	0.0	11.0
		30	8.07	0.0	0.0	0.0	11.0
		10	7.78	0.1	0.0	0.0	7.0
	512	20	7.52	0.0	0.0	0.0	7.0
L		30	6.73	0.0	0.0	0.0	7.0
		10	7.68	0.0	0.0	0.0	7.0
	1024	20	7.55	0.1	0.0	0.0	7.0
H		30	11.78	80.6	0.0	0.0	7.0

Analysis: Excluding the 1024/30 combination, it appears that having to write 7 pages to disk instead of 11 improves the execution time slightly. The 1024/30 case looks like an oddity since it has so many system page faults, when compared to the other combinations. For whatever reason they occurred, the extra page faults probably explain the larger execution times.

In a new spatial database, all pages created by the DB_CREATE command are dirty. As expected, these pages are written to disk when the database is closed.

The characteristics for a database are not affected by the DB_DOWN command.

Test 3

Bringing up an Existing Database

Test: Determine the average amount of time required to bring up an existing database using the DB_UP command. This process involves initializing the page buffers and accessing the 4 system relations. The only relationship between the speed of DB_UP and the database size is the time required to locate the relation's name and TID in the $$RELCATALOGRELNAME B-tree. Like DB_CREATE, DB_UP reinitializes the page buffers each time the command is executed.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	699.48	0.5	1.0	11.0	0.0
	256	20	700.67	0.7	0.0	11.0	0.0
		30	701.07	0.7	0.0	11.0	0.0
L		10	452.81	0.2	0.0	7.0	0.0
	512	20	699.21	0.2	0.0	7.0	0.0
		30	701.70	0.1	0.0	7.0	0.0
		10	699.59	0.6	0.0	7.0	0.0
	1024	20	700.34	0.3	0.0	7.0	0.0
H		30	718.52	272.2	0.0	7.0	0.0

Analysis: As with Test 1, the 512/10 and 1024/30 combinations are the exceptions in fairly consistent data. Note that all pages created by the DB_CREATE command (Test 1) had to be read in.

This is because DB_UP, like DB_CREATE, creates an Active_Relationlist entry for the RELCATALOG, $$RELCATALOGRELNAME, and BTREETIDLIST relations - a process which involves reading in all 4 system relations (LINKNODEPROTO is read in since it is the prototype for the BTREETIDLIST relation).

The characteristics of a database are not affected by the DB_UP command.

Test 4

Closing an Modified Database

Test: Determine the average amount of time required to write a unmodified spatial database that is up to disk using the DB_DOWN command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	4.10	0.1	0.0	0.0	1.0
	256	20	4.23	0.1	0.0	0.0	1.0
L		30	3.84	0.1	0.0	0.0	1.0
		10	4.38	0.0	0.0	0.0	1.0
	512	20	4.49	0.0	0.0	0.0	1.0
		30	4.23	0.0	0.0	0.0	1.0
		10	4.58	0.0	0.0	0.0	1.0
	1024	20	4.82	0.0	0.0	0.0	1.0
H		30	7.72	67.4	0.0	0.0	1.0

Analysis: Other than 1024/30, all combinations of page size and number of page buffers appear fairly even. The smaller 256-byte page size caused more system page faults, but this did not affect the overall execution time. Since none of the system relations were changed while the database was up, only page 1 (system constants) were written to disk. As with the earlier tests, perhaps the fact that 30 1024-byte pages produced a large database system program explains why the 1024/30 combination had an excessive number of system page

faults. There must be some reason since the 1024/30 combination has been the worst in the 4 tests run so far.

Closing a database involves releasing the Active_Relationlist entries for the system relations, a process which necessitates rereading the RELCATALOG tuples for the system relations (hopefully already in the page buffers) and determining that no changes took place.

The characteristics of the databases are not modified if DB_DOWN is issued and no changes occurred.

Comparing these results with those from Test 2 (Closing a new database), one can see that it takes approximately twice as long to write the new database to disk. This difference is easily attributed to having to perform the extra writes for pages containing the system relations.

Test 5

Creating a Relational Prototype for an Unordered Relation

Test: Determine the average amount of time required to create a relational prototype for an unordered relation using the CREATE_RELPROTO command. The length of tuples defined by the relational prototype will be 0 since no fields have been defined.

Results:

Performance Measurement Statistics - For One Command

Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
	10	288.26	3.3	1.5	0.1	1.3
256	20	292.14	2.4	1.1	0.0	1.1
	30	293.14	1.6	1.0	0.0	1.0
L	10	210.96	0.5	0.3	0.0	0.3
512	20	318.72	0.5	0.1	0.0	0.1
	30	330.40	0.5	0.0	0.0	0.0
	10	489.20	0.5	0.1	0.0	0.0
1024	20	491.08	0.4	0.0	0.0	0.0
H	30	513.20	415.5	0.0	0.0	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+36	+22	+20
# Pages used	+71	+21	+9
# Tuples	+117	+107	+103
# Relations	+50	+50	+50
# Prototypes	+50	+50	+50
# SDSs	+0	+0	+0

Analysis: The execution time does not seem to be influenced by the number of system page faults, except in the 1024/30 case, which appears to be an oddity. For all page sizes, more buffers means slower execution times, fewer database page faults, and fewer writes. There is also some relationship between page size and execution time, except for the 512/10 case; larger pages means slower execution times.

The process for creating a relational prototype involves creating a tuple for the prototype in the Relational Catalog and the catalog's accessing B-tree, $$RELCATALOGRELNAME. Since the prototypes created have similar names, the Relational Catalog B-tree was rebalanced quite frequently.

The number of tuples created include 50 RELCATALOG tuples, plus however many tuples were required by the $$RELCATALOGRELNAME B-tree. The relational prototypes are all empty (i.e. no fields are defined), so all additional page space was used for the Relational Catalog tuples. Having 1024-byte pages reduces the number of pages used to represent the database, but, as comparing the 512-byte and 1024-byte statistics shows, the larger page size does not

greatly improve the number of VMS blocks used by the database (Recall one 1024-byte page = two VMS 512-byte blocks).

Test 6

Creating a Relational Prototype for an Ordered Relation

Test: Determine the average amount of time required to create a relational prototype for an ordered relation using the CREATE_RELPROTO command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	502.30	285.9	2.8	0.5	2.7
	256	20	502.56	260.7	2.2	0.0	2.2
L		30	499.12	207.3	2.0	0.0	2.0
		10	533.46	248.4	1.4	0.1	1.4
	512	20	533.96	249.2	1.1	0.0	1.1
		30	534.76	272.3	0.9	0.0	0.9
		10	708.92	375.2	1.1	0.0	1.1
	1024	20	711.84	399.5	0.9	0.0	0.9
H		30	712.62	359.2	0.7	0.0	0.7

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+61	+72	+120
# Pages used	+121	+71	+59
# Tuples	+167	+147	+153
# Relations	+50	+50	+50
# Prototypes	+50	+50	+50
# SDSs	+0	+0	+0

Analysis: This test is the first test to create relations whose results did not favor the 512/10 combination. In general, smaller page sizes faired better, in spite of the fact that more database page faults and writes occurred.

The process for creating a relational prototype involves creating a tuple for the prototype in the Relational Catalog and the catalog's accessing B-tree, $$RELCATALOGRELNAME. Since the prototypes created have similar names, the B-tree was rebalanced quite frequently. Because the prototype defines ordered relations, a field definition for a NEXTTID field was automatically added to the prototype.

The number of tuples created includes 50 RELCATALOG tuples, $$RELCATALOGRELNAME B-tree tuples, and 1 tuple per prototype for the NEXTTID field. Note that relational prototypes are counted as relations.

The NEXTTID tuple occupies one page in <u>each</u> relational, so the number of pages (and blocks) used by the database is greatly increased from when compared to Test 5 (same test for unordered relations). The execution times for ordered relations were much slower than their unordered counterparts (Test 5 again), due to the fact that processing for an ordered relation must access each page to insert the NEXTTID field, while processing for each unordered relation does not even have to allocates pages to the relation.

Test 7

Add a Field to Many Relational Prototypes for Unordered Relations

Test: Determine the average amount of time required by the ADDTO_RELPROTO command to add the definition of a single 15-byte character field to multiple relational prototypes. The prototypes all define unordered relations.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	205.26	95.6	2.5	1.4	1.9
	256	20	204.18	99.3	2.4	1.3	1.7
		30	204.54	107.0	2.1	1.4	1.5
L		10	127.86	0.7	1.4	0.4	1.1
	512	20	197.30	2.9	1.2	0.4	1.0
		30	197.24	0.9	1.0	0.4	0.8
		10	197.06	0.2	1.2	0.2	1.0
	1024	20	197.38	0.7	0.9	0.2	0.8
		30	204.30	132.2	0.8	0.2	0.6

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+25	+50	+100
# Pages used	+50	+50	+50
# Tuples	+50	+50	+50
# Relations	+0	+0	+50
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: Other than the 512/10 case, there do not appear to be major differences between how the various combinations performed. Although the number of system page faults varied, the only trend seemed to be that the smaller 256-byte page produced many more system page faults, resulting in a slight increase in the execution time. Fewer page buffers did produce additional database page faults, as expected. In all cases, database reads and writes occurred as the Relational Catalog information for all 50 relations was updated, in addition to inserting one field in one page of each relation (Note-adding a new page to a relation only causes a write).

One tuple, occupying one page, was added to <u>each</u> relational prototype. Note that in this case, the relationship between the number of blocks used and the number of pages used was exactly what was expected (i.e. two 256-byte pages = one 512-byte page = 1/2 1024-byte page = one 512-byte VMS block). This means that when a 1024-byte page was used, two VMS blocks were allocated for the record, even though most of the record was empty.

Test 8

Add a Field to Many Relational Prototypes for Ordered Relations

Test: Determine the average amount of time required by the ADDTO_RELPROTO command to add the definition of a single 15-byte character field to multiple relational prototypes. The prototypes all define ordered relations.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	207.76	123.5	2.4	2.4	1.9
	256	20	205.90	121.5	2.3	2.4	1.7
		30	207.84	149.4	2.1	2.4	1.6
L		10	204.00	101.9	1.4	1.4	1.1
H	512	20	209.80	162.9	1.2	1.5	1.0
		30	207.10	119.0	1.0	1.4	0.8
		10	207.74	152.9	1.2	1.2	1.0
	1024	20	207.28	118.4	1.0	1.2	0.9
		30	205.06	129.5	0.8	1.2	0.6

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+0	+0	+0
# Pages used	+0	+0	+0
# Tuples	+50	+50	+50
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: There are no significant differences between the statistics observed for the various page size/# buffer combinations. More page buffers resulted in fewer database page faults, a trend that is expected since each relational prototype accessed involved a unique set of pages.

With this test, there are two tuples in each relational prototype, since a NEXTTID tuple was added when the prototype was created. Both tuples reside on the same page so no new pages or blocks were needed.

Unordered relations (Test 7) performed slightly better with fewer database page faults and reads. This is likely because when a relation is accessed, a read is automatically done for the first tuple in the relation. For new ordered relations, this is the NEXTTID tuple; for unordered relations, no read takes place since there are no tuples in the relation. The number of system page faults also tended to be similar for unordered relations. The number of writes performed is consistent since the updated page (containing 1 tuple for unordered relations, 2 for ordered relations), must still be written to disk for ordered and unordered relations.

Test 9

Listing the Catalog Information for a Relation

Test: Determine the average amount of time required by the LISTREL command to list the information in the Relational Catalog specific to a single relation. The type of relation used in this test is unimportant since all relations have a similar tuple in the Relational Catalog.

Results:

Performance Measurement Statistics - For One Commands

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	67.24	0.3	1.4	1.4	0.0
	256	20	67.30	0.1	1.3	1.4	0.0
		30	67.36	1.0	1.0	1.4	0.0
L		10	43.90	0.5	0.4	0.4	0.0
	512	20	67.18	0.4	0.2	0.4	0.0
		30	67.04	0.2	0.0	0.4	0.0
		10	67.12	0.6	0.1	0.2	0.0
	1024	20	67.10	0.9	0.0	0.2	0.0
H		30	71.28	69.3	0.8	1.2	0.0

Analysis: Other than the 512/10 and 1024/30 combinations, there do not seem to be significant differences between the various combinations tested. The 1024/30 combination had quite a few more system page faults (perhaps because 1024/30 results in the largest program size), but that increase did not produce a comparable gain in the execution time. The good performance of the 512/10 combination is consistent

with the observations of earlier tests, although the source of this improvement is still unknown.

The only database accesses were to search the $$RELCATALOGRELNAME B-tree for the specified relation name, and to finally read in the RELCATALOG tuple identified by the TID in the B-tree. With smaller page sizes, fewer B-tree keys (in this case, the relation name) fit on each page and more database accesses were required to traverse the B-tree.

The characteristics of the database are not changed by the LISTREL command.

Test 10

Creating an Unordered Relation

Test: Determine the average amount of time required by the CREATEREL command to create an unordered relation. The relation is defined by its prototype to have one 15-byte character field that is not indexed. Since no fields in the relation are indexed, an accessing B-tree will be created and subsequent tuples in the relation will be ordered by their tuple identifier (TID).

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	944.74	587.7	20.5	17.8	10.4
L	256	20	931.80	486.6	6.5	3.7	3.6
		30	934.12	492.3	5.0	2.5	2.6
		10	1037.86	533.7	8.7	7.8	3.9
	512	20	1030.50	532.3	2.3	1.6	0.9
		30	1029.20	503.2	2.0	1.3	0.7
		10	1284.36	670.3	3.6	3.2	1.3
	1024	20	1283.82	681.2	1.5	1.3	0.5
H		30	1286.00	810.9	1.3	1.3	0.3

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+72	+46	+44
# Pages used	+142	+46	+22
# Tuples	+233	+217	+207
# Relations	+100	+100	+100
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: In general, larger page sizes meant slower execution times, fewer database page faults, fewer reads, and fewer writes. More page buffers resulted in fewer database accesses and page faults, but this did not appear to affect the overall execution time.

The creation process for both the unordered relation and the accessing B-tree (also a relation) requires the database system to read in all of the Relational Catalog information for each relation's prototype, as well as the entire prototype definition. New RELCATALOG tuples are then created for the unordered relation and its B-tree. Because the names of the relations created were similar, the B-tree for the Relational Catalog was balanced frequently.

The only observable trend in the database characteristics. is that smaller pages meant more system blocks and pages. This difference was insignificant when page sizes larger than 256 bytes were used. The 100 relations created include 50 B-trees and 50 unordered relations. All of the increase in the number of tuples can be attributed to RELCATALOG and

$$RELCATALOGRELNAME entries. There are no tuples in any of the 50 unordered relations created.

Test 11

Creating an Ordered Relation

Test: Determine the average amount of time required to create an ordered relation using the CREATEREL command. The relation is defined by its prototype to have one 15-byte. A 4-byte NEXTTID field will be present in each tuple of the relation.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	733.84	380.9	10.3	8.9	4.2
	256	20	727.70	344.1	4.0	2.7	1.5
		30	727.22	336.8	3.6	2.5	1.3
		10	712.36	373.9	4.4	4.4	1.7
L	512	20	707.94	378.4	1.8	1.6	0.5
		30	708.30	364.9	1.5	1.5	0.3
		10	759.98	384.6	2.0	1.8	0.5
	1024	20	759.56	386.8	1.2	1.3	0.2
H		30	762.72	450.6	1.0	1.2	0.1

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+36	+23	+70
# Pages used	+71	+23	+10
# Tuples	+116	+109	+103
# Relations	+50	+50	+50
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: In general, a page size of 512 resulted in better average execution times, perhaps because 512 is the page size used by VMS. More page buffers yielded fewer database page faults, reads, and writes.

The creation process for the ordered relation requires the database system to read in all of the Relational Catalog information for each relation's prototype, as well as the entire prototype definition. New RELCATALOG tuples are then created for the ordered relation. Because the names of the relations created were similar, the B-tree for the Relational Catalog was balanced frequently. There is no accessing B-tree for ordered relations.

The increase in the number of tuples can be attributed to the RELCATALOG and $$RELCATALOGRELNAME entries, plus one NEXTTID field for each of the 50 relations.

The differences between Tests 10 (CREATEREL for unordered relations) and 11 lead one to believe that ordered relations should be created, even if tuple ordering does not matter. Unordered relations lost ground because an accessing B-tree had to be created for each unordered relation created. Note that an opposite conclusion could be drawn from the Test 5 and 6 results, which observed that creating a relational prototype for an ordered relation took more time since a NEXTTID field had to be added to the prototype relation. Since many relations are likely to be created from one relational prototype, the earlier results are less significant than those from Test 10 and 11. Ordered relations also use less database space than unordered relations since no additional pages are required to support an accessing B-tree.

Test 12

Determine the Name of a Relation's Prototype

Test: Determine the average amount of time required to determine the name of a relation's prototype using the PROTOTYPE command. The type of relation used in this test is unimportant since the PROTOTYPE command accesses a relation's tuple in the Relational Catalog, not the actual relation itself.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	24.00	8.6	1.5	1.5	0.0
	256	20	24.16	8.5	1.3	1.4	0.0
H		30	24.40	18.6	1.1	1.4	0.0
		10	23.56	5.2	0.5	0.5	0.0
L	512	20	23.50	7.7	0.3	0.5	0.0
		30	23.44	4.1	0.1	0.5	0.0
		10	23.88	12.6	0.2	0.3	0.0
	1024	20	23.54	7.8	0.0	0.3	0.0
		30	23.66	7.8	0.0	0.3	0.0

Analysis: There do not seem to be significant differences between the various page size/# buffers combinations tested. The number of system page faults varied, but this did not appear to affect average execution times. The 256 page size combinations had more database page faults, resulting in slightly higher execution times.

The bulk of the execution time was likely spent searching the Relational Catalog B-tree for the desired relation name (which now contained 54 keys), and then accessing the appropriate tuple in the Relational Catalog.

The characteristics of a database are not changed by the PROTOTYPE command.

Test 13

Add a Tuple to an Unordered Relation

Test: Determine the average amount of time required to add a new tuple consisting of a 15-byte character field to an unordered relation using the BUILD_TUPLE command. Since the relation is not ordered and no fields are indexed, the TID for the new tuple will also be added to the accessing B-tree for the relation.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	133.64	77.4	0.4	0.0	0.2
	256	20	132.72	58.4	0.4	0.0	0.0
		30	133.40	75.0	0.3	0.0	0.0
		10	128.64	70.9	0.2	0.0	0.0
	512	20	127.34	77.4	0.1	0.0	0.0
		30	127.26	74.7	0.0	0.0	0.0
L		10	110.14	48.6	0.1	0.0	0.0
	1024	20	112.46	44.2	0.0	0.0	0.0
		30	111.84	45.1	0.0	0.0	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+10	+10	+8
# Pages used	+19	+9	+4
# Tuples	+108	+104	+101
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: In this test, the page size is directly related to the number of system page faults recorded. With a larger page size, fewer system faults occur, resulting in smaller average execution times. Because the Relational Catalog information for the unordered relation must already be in memory prior to issuing a BUILD_TUPLE command, a large proportion of the execution time was spent updating the relation's accessing B-tree. The BUILD_TUPLE command will not cause the new tuple to be written to disk, unless forced to by a subsequent database page fault.

The increases in the database characteristics include one tuple for the unordered relation, plus however many tuples were needed to update the relation's accessing B-tree.

Test 14

Add a Tuple to an Ordered Relation

Test: Determine the average amount of time required by the BUILD_TUPLE command to add a new tuple consisting of a 15-byte character field to an ordered relation.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	46.00	32.8	0.1	0.0	0.0
L	256	20	44.92	12.3	0.1	0.0	0.0
H		30	46.76	36.8	0.0	0.0	0.0
		10	45.38	26.1	0.1	0.0	0.0
	512	20	45.96	25.4	0.0	0.0	0.0
		30	45.86	35.7	0.0	0.0	0.0
		10	45.26	11.0	0.0	0.0	0.0
	1024	20	46.16	34.0	0.0	0.0	0.0
		30	45.74	31.3	0.0	0.0	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+4	+3	+4
# Pages used	+7	+3	+2
# Tuples	+50	+50	+50
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: There are no significant differences between the various execution times recorded, or between the execution time and the number of system page faults. The 1024/10 and 256/20 combinations had few system page faults, but their execution times are not comparably better than the other combinations tested.

Because the Relational Catalog information for the ordered relation must already be in memory prior to issuing the BUILD_TUPLE command, the bulk of the execution time was spent linking this tuple to its predecessor in the relation. BUILD_TUPLE will not cause the new tuple to be written to disk, unless forced to by a subsequent database page fault.

Comparing the Test 14 results with those from Test 13 (BUILD_TUPLE for unordered relations), one can see the difference having to access a second relation (the unordered relation's B-tree) can make. Ordered relations resulted in fewer increases in the database characteristics, again because an accessing B-tree did not have to be updated or rebalanced.

Test 15

Accessing the Next Tuple in an Unordered Relation

Test: Determine the average amount of time required by the NEXT_TUPLE command to access the next tuple in an unordered relation. Since the tuples in the relation do not have a NEXTTID field, the relation's accessing B-tree will be used to determine which tuple succeeds the tuple currently in memory.

Results:

Performance Measurement Statistics

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	0.82	0.8	0.2	0.2	0.0
H	256	20	0.84	0.8	0.2	0.2	0.0
		30	0.80	0.8	0.1	0.2	0.0
		10	0.72	0.2	0.1	0.1	0.0
H	512	20	0.84	1.1	0.0	0.1	0.0
		30	0.82	1.3	0.0	0.1	0.0
		10	0.70	0.8	0.0	0.0	0.0
	1024	20	0.70	1.3	0.0	0.0	0.0
L		30	0.64	0.5	0.0	0.0	0.0

Analysis: The execution times of the 1024 combinations were better than the other variations, perhaps because the larger page size resulted in the "next tuple" being on a page already in memory. The number of system page faults did not appear related to the execution time.

Note that two database accesses are required per NEXT_TUPLE command: one access to retrieve the TID of the next tuple from the accessing B-tree, and one access to retrieve the actual tuple.

The characteristics of the database are not changed by the NEXT_TUPLE command.

Test 16

Accessing the Next Tuple in an Ordered Relation

Test: Determine the average amount of time required by the NEXT_TUPLE command to access the next tuple in an ordered relation. The NEXTTID field of the current tuple will be used to determine which tuple to read in next; ordered relations have no accessing B-tree.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	0.78	1.8	0.2	0.1	0.0
	256	20	0.74	0.6	0.1	0.1	0.0
		30	0.58	0.4	0.0	0.1	0.0
		10	0.64	1.4	0.0	0.0	0.0
L	512	20	0.48	0.1	0.0	0.0	0.0
		30	0.60	0.5	0.0	0.0	0.0
		10	0.62	1.3	0.0	0.0	0.0
	1024	20	0.54	0.1	0.0	0.0	0.0
		30	0.72	0.9	0.0	0.0	0.0

Analysis: With page sizes larger than 256 bytes, all database statistics went to zero, implying that database I/O did not contribute to the overall execution time. System page faults still occurred, however. Because such a potpourri of execution times were recorded, no conclusions about the relationships between execution time, page size, and number of page buffers can be drawn.

The characteristics of the database are not changed by the NEXT_TUPLE command.

When compared to the same command for unordered relations (Test 15), one can see the improvement in execution time obtained by not having to access a B-tree.

Test 17

Retrieving the Value of a Field in a Relation

Test: Determine the average amount of time needed to retrieve the value of a specific field in a tuple using the VALUE_OF command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	22.58	8.8	0.0	0.0	0.0
	256	20	22.68	13.9	0.0	0.0	0.0
		30	22.90	19.8	0.0	0.0	0.0
L		10	22.46	8.6	0.0	0.0	0.0
	512	20	22.68	13.2	0.0	0.0	0.0
		30	22.80	17.3	0.0	0.0	0.0
		10	22.76	16.1	0.0	0.0	0.0
	1024	20	22.80	18.6	0.0	0.0	0.0
H		30	22.98	12.3	0.0	0.0	0.0

Analysis: There do not appear to be significant differences between the results obtained for the various page size/# buffers combinations. Although 256/10 and 512/10 had fewer system page faults, their average execution times were only slightly better than the other combinations. Since the tuple with the field being retrieved was already in memory, no database reads took place.

The characteristics of the database are not changed by the VALUE_OF command.

Test 18

Setting a Non-Indexed Field in a Tuple

Test: Determine the average amount of time required to set an non-indexed 15-byte character field in a tuple using the SETVALUE command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	22.90	15.6	0.0	0.0	0.0
	256	20	22.80	19.3	0.0	0.0	0.0
L		30	20.88	21.9	0.0	0.0	0.0
		10	22.78	19.7	0.0	0.0	0.0
	512	20	22.50	1.0	0.0	0.0	0.0
		30	22.50	13.6	0.0	0.0	0.0
H		10	22.94	23.9	0.0	0.0	0.0
	1024	20	22.86	20.4	0.0	0.0	0.0
		30	22.64	14.2	0.0	0.0	0.0

Analysis: The 256/30 combination faired the best, although it appears to be an oddity when compared to the other statistics. There do not seem to be any relationships between the page sizes, number of buffers, and number of system page faults (consider 512/20 versus 256/30). Since the tuple being updated must be in memory prior to issuing the SETVALUE command, no database reads were required.

The characteristics of the database are not changed by the SETVALUE command.

Test 19

Setting an Indexed Field in a Tuple

Test: Determine the average amount of time required by the SETVALUE command to set an indexed 15-byte character field in an unordered relation. Note that this test cannot be run for ordered relations since they cannot have indexed fields.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	342.54	121.3	5.0	4.9	4.3
	256	20	342.84	111.9	0.8	0.2	0.4
L		30	338.08	123.6	0.4	0.4	0.0
		10	562.38	223.5	0.7	0.6	0.6
	512	20	567.14	253.6	0.2	0.2	0.0
		30	561.00	235.4	0.1	1.6	0.0
H		10	878.04	438.8	1.2	1.2	0.0
	1024	20	876.44	394.6	0.0	0.1	0.0
		30	875.30	388.1	0.0	0.2	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+2	+2	+0
# Pages used	+3	+2	+2
# Tuples	+3	+2	+2
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: The execution time is directly related to the page size. Larger pages mean more system page faults and slower execution times. Perhaps this discrepancy is due to the distribution of keys on the various B-tree pages, and the resulting need for tree rebalancing after both old key deletion and new key insertion. The number of page buffers used does not appear to affect the overall execution time, even though more database initiated I/O was recorded with fewer page buffers.

There do not appear to be any significant differences between the recorded database characteristics. Insertion of the new field values caused 2-3 more database pages and tuples to be used, most likely due to the change in the number of keys in each B-tree node and any tree rebalancing that took place.

Comparing the test results from this test with those from Test 18 (same command, non-indexed field) shows the impact on database performance of specifying a field as INDEXED - disk accesses are required to delete the old field value, insert the new field value, and rebalance the B-tree.

Test 20

Print a Tuple in an Unordered Relation

Test: Determine the average amount of time required to print a tuple in an unordered relation using the PRINT_TUPLE command. The indexing type of each field is unimportant since the tuple must already be in memory prior to issuing this command.

Results:

Performance Measurement Statistics - For One Command

Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
	10	23.32	13.5	0.0	0.0	0.0
256	20	22.86	10.9	0.0	0.0	0.0
	30	22.68	8.2	0.0	0.0	0.0
	10	22.76	7.0	0.0	0.0	0.0
512	20	23.08	17.4	0.0	0.0	0.0
H	30	23.36	16.0	0.0	0.0	0.0
	10	22.76	8.3	0.0	0.0	0.0
1024	20	22.64	14.1	0.0	0.0	0.0
L	30	22.52	7.5	0.0	0.0	0.0

Analysis: The only conclusion that can be drawn from the statistics is that there appears to be a slight relationship between the execution time and the number of system page faults. More page faults yielded slower execution times. The number of database reads and page faults was not zero (the pages had to get in somehow), but there were fewer than 2.5 reads for the 50 tuples printed. It is expected that the bulk of the execution time can be attributed to the time

required to write to the operator's console, and not to the few database reads.

The characteristics of the database are not changed by the PRINT_TUPLE command.

Test 21

Print a Tuple in an Ordered Relation

Test: Determine the average amount of time required to print a tuple in an ordered relation using the PRINT_TUPLE command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	22.88	6.3	0.0	0.0	0.0
H	256	20	23.20	16.3	0.0	0.0	0.0
		30	22.96	11.4	0.0	0.0	0.0
		10	22.80	6.5	0.0	0.0	0.0
L	512	20	23.12	14.8	0.0	0.0	0.0
		30	23.08	16.2	0.0	0.0	0.0
		10	22.84	9.1	0.0	0.0	0.0
L	1024	20	22.80	9.5	0.0	0.0	0.0
		30	22.90	12.3	0.0	0.0	0.0

Analysis: There do not appear to be significant differences between execution times of the various page size/# buffer combinations, in spite of the variations in the number of system page faults. In all cases, more system page faults generally resulted in slower execution times.

The characteristics of the database are not changed by the PRINT_TUPLE command.

There are no significant differences between the results observed from this test and values obtained for Unordered

Relations in Test 20. This is not much of a surprise since the tuple must already be in memory prior to issuing the PRINT_TUPLE command. The only difference between printing unordered and ordered relations is the NEXTTID field which must be printed for each tuple in an ordered relation.

Test 22

Access & Release an Unordered Relation - No Changes

Test: Determine the average amount of time required to create (ACCESSREL command) and release (RELEASEREL command) an Active_Relationlist (ARL) entry containing the Relational Catalog and prototype information for an unordered relation.

Results:

Performance Measurement Statistics - ACCESSREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	394.36	207.8	9.9	9.6	0.0
	256	20	383.94	166.7	5.0	5.0	0.0
		30	384.72	198.5	4.8	5.0	0.0
		10	386.72	231.9	3.0	3.0	0.0
	512	20	389.46	194.0	2.1	2.3	0.0
L		30	382.76	193.0	1.9	2.3	0.0
		10	383.04	208.3	1.6	1.7	0.0
	1024	20	391.00	203.1	1.4	1.7	0.0
		30	387.64	194.8	1.2	1.7	0.0

Performance Measurement Statistics - RELEASEREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	24.28	15.6	0.0	0.0	0.0
	256	20	23.80	8.9	0.0	0.0	0.0
		30	23.44	11.0	0.0	0.0	0.0
		10	23.60	17.7	0.0	0.0	0.0
H	512	20	24.80	4.3	0.0	0.0	0.0
		30	23.52	17.4	0.0	0.0	0.0
L		10	23.30	19.9	0.0	0.0	0.0
	1024	20	23.50	15.8	0.0	0.0	0.0
		30	23.50	19.0	0.0	0.0	0.0

Analysis: For ACCESSREL, there do not appear to be any trends in the execution time, in spite of the observation that more buffers produced lower values for the other measured statistics. The actions taken in creating an ARL entry are similar for all page sizes: 1) search the $$RELCATALOGRELNAME B-tree for the TID of the tuple in the RELCATALOG representing the relation, 2) read in the RELCATALOG tuple for the relation, 3) read in the entire prototype for the relation, 4) access the B-tree (also an ACCESSREL command) for the unordered relation, and 5) read in the first tuple for the relation, if one exists. Given the number of database accesses involved, one would expect that having larger page buffers would yield better execution times since fewer database reads would be required. This hypothesis was not completely correct; more and larger page buffers did result in fewer database accesses, but the fewer accesses did not greatly affect the execution time. Perhaps this is because the bulk of the

processing for each relation accessed involved references to the RELCATALOG and $$RELCATALOGRELNAME relations, which were likely kept in memory by the LRU paging algorithm.

Like ACCESSREL, no trends manifested themselves in the RELEASEREL results. For all page sizes, RELEASEREL involves reaccessing the RELCATALOG tuple for the relation being released, and determining that the tuple does not have to be modified since no changes occurred.

The characteristics of the database are not changed by the ACCESSREL or RELEASEREL commands.

Test 23

Access & Release an Ordered Relation - No Changes

Test: Determine the average amount of time required to create (ACCESSREL command) and release (RELEASEREL command) an Active_Relationlist (ARL) entry containing the Relational Catalog and prototype information for an ordered relation.

Results:

Performance Measurement Statistics - ACCESSREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	428.82	206.7	5.4	5.4	0.0
L	256	20	428.32	209.6	3.8	3.9	0.0
		30	428.56	225.9	3.6	3.9	0.0
		10	428.70	231.7	2.7	2.7	0.0
	512	20	428.38	237.3	1.7	2.0	0.0
		30	428.40	236.2	1.6	1.9	0.0
		10	428.76	236.3	1.5	1.5	0.0
	1024	20	428.62	236.8	1.3	1.5	0.0
L		30	428.32	213.5	1.0	1.5	0.0

Performance Measurement Statistics - RELEASEREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	23.78	24.8	0.0	0.0	0.0
	256	20	23.50	19.8	0.0	0.0	0.0
		30	23.60	17.1	0.0	0.0	0.0
		10	23.60	19.3	0.0	0.0	0.0
	512	20	23.58	20.8	0.0	0.0	0.0
		30	23.50	19.7	0.0	0.0	0.0
L		10	23.44	17.5	0.0	0.0	0.0
	1024	20	23.66	19.5	0.0	0.0	0.0
		30	23.60	17.6	0.0	0.0	0.0

Analysis: For both ACCESSREL and RELEASEREL, there do not appear to be significant differences or relationships between the values page size/# buffer combinations. Refer to the analysis of Test 22 (same combination for unordered relations) for some possible explanations why.

The characteristics of the database are not changed by the ACCESSREL or RELEASEREL commands.

There are differences with the results obtained in Test 22 (ACCESSREL and RELEASEREL for unordered relations). In general, accessing and releasing an ordered relation is faster since there is no additional B-tree to access and release.

Test 24

Release an Unordered Relation - With Changes

Test: Determine the average amount of time required to release (RELEASEREL command) an Active_Relationlist (ARL) entry containing the RELCATALOG information for an unordered relation whose catalog information has changed. No statistics will be recorded for the ACCESSREL command (create an ARL entry) with changes since this command's performance is not influenced by subsequent uses of the ARL.

Results:

Performance Measurement Statistics - RELEASEREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	23.78	20.1	0.0	0.0	0.0
	256	20	23.50	10.1	0.0	0.0	0.0
L		30	23.16	5.1	0.0	0.0	0.0
		10	23.90	17.1	0.0	0.0	0.0
	512	20	23.70	10.8	0.0	0.0	0.0
H		30	24.84	35.3	0.0	0.0	0.0
		10	23.44	11.3	0.0	0.0	0.0
	1024	20	23.42	11.8	0.0	0.0	0.0
		30	23.56	10.7	0.0	0.0	0.0

Analysis: There do not appear to be any differences between the various combinations of RELEASEREL tested, although more system page faults generally meant a slight increase in the execution time.

The characteristics of a database are not changed by the RELEASEREL command.

There are no major differences between the results from this test and those from Test 22 (same command without ARL changes). This is likely because RELEASEREL causes the RELCATALOG tuple for the relation being released to be reaccessed to determine whether any changes occurred. This means that the page is always accessed, even if the comparisons eventually determine that no changes took place.

Test 25

Release an Ordered Relation - With Changes

Test: Determine the average amount of time required to release (RELEASEREL command) an Active_Relationlist (ARL) entry containing the RELCATALOG information for an ordered relation whose catalog information has changed.

Results:

Performance Measurement Statistics - RELEASEREL

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	23.30	10.8	0.0	0.0	0.0
L	256	20	22.90	12.4	0.0	0.0	0.0
H		30	27.00	9.4	0.0	0.0	0.0
		10	23.32	19.1	0.0	0.0	0.0
	512	20	23.48	26.8	0.0	0.0	0.0
		30	23.66	16.6	0.0	0.0	0.0
		10	23.18	9.5	0.0	0.0	0.0
	1024	20	23.54	13.5	0.0	0.0	0.0
		30	23.14	8.2	0.0	0.0	0.0

Analysis: Other than the 256/30 case, there do not appear to be any differences between the various combinations of RELEASEREL tested, although more system page faults generally mean a slight increase in the execution time.

The characteristics of a database are not changed by the RELEASEREL command.

There are no major differences between these results and those obtained from Test 23 (same command without ARL changes). This is likely because RELEASEREL causes the RELCATALOG tuple for the relation being released to be reaccessed to determine whether any changes occurred. This means that the page is always accessed, even if the comparisons eventually determine that no changes took place.

As observed in Test 23-25, the RELEASEREL command for an ordered relation executes slightly faster than the same command for unordered relations since no accessing B-tree has to be released.

Test 26

Print an Ordered Relation with One Field

Test: Determine the average amount of time required by the PRINTREL command to print an ordered relation containing 50 tuples, each with one non-indexed field. The NEXTTID field in each tuple will be used to determine the tuple's successor.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	1521.08	1042.7	17.0	17.0	0.0
L	256	20	1508.26	961.1	0.2	0.3	0.0
		30	1524.34	1364.3	0.3	0.3	0.0
		10	1520.88	1035.7	10.0	10.0	0.0
	512	20	1511.74	1030.4	0.2	0.0	0.0
		30	1527.80	1082.1	0.2	0.0	0.0
		10	1523.98	1370.5	0.1	0.2	0.0
H	1024	20	1549.78	1067.1	0.0	0.2	0.0
		30	1508.66	954.9	0.0	0.2	0.0

Analysis: In general, having only 10 page buffers resulted in more database page faults. This is because the 50 tuples in the relations printed occupied more than 10 pages. The 1024/30 and 256/20 combinations executed the fastest because less system I/O was performed. The only I/O required by PRINTREL was to read the next page of a relation once all tuples on the current page have been printed. This is because

successive tuples in a new relation occupy contiguous pages or the same pages.

In general, the database system performed about the same for all page size/# buffer combinations. There does not appear to be a relationship between the number of page buffers, the page size, and the execution time.

Note that page fragmentation will result in more database and system page faults (and perhaps a slower PRINTREL command) if tuples are deleted and inserted. This is because there is no guarantee that tuples follow each other will be clustered on the same pages.

Processing for the PRINTREL command includes commands that have already been tested: ACCESSREL, PRINT_TUPLE and NEXT_TUPLE for each tuple, and RELEASEREL. Given that knowledge, it is no major challenge to predict the time required to print an ordered relation with a specified number of tuples. The actual printing time will be slightly less since fewer interpreter integrity checks will be performed for PRINTREL.

The characteristics of a database are not affected by the PRINTREL command.

Test 27

Print an Unordered Relation with One Field

Test: Determine the average amount of time required by the PRINTREL command to print unordered relation containing 50 tuples, each with one non-indexed field. The accessing B-tree for the relation will order each tuple in the relation by its TID.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	1464.70	893.7	24.0	24.0	0.0
	256	20	1468.18	887.9	22.9	22.9	0.0
		30	1470.42	1278.8	0.1	0.4	0.0
		10	1479.20	891.5	13.0	13.0	0.0
	512	20	1483.78	1301.5	0.0	0.2	0.0
		30	1464.00	992.3	0.0	0.2	0.0
L		10	1463.22	944.1	8.0	8.0	0.0
H	1024	20	1489.72	1332.1	0.0	0.2	0.0
		30	1471.10	929.2	0.0	0.2	0.0

Analysis: The relationship between the execution time and the number of system page faults appears to be that fewer system page faults resulted in better execution times. Fewer page buffers did yield more database page faults, but this did not cause the 10 page buffer combinations to perform any worse than their 30 buffer counterparts.

Processing for the PRINTREL command includes commands that have already been tested: ACCESSREL, PRINT_TUPLE and NEXT_TUPLE for each tuple, and RELEASEREL. Given that knowledge, it is no major challenge to predict the time required to print an unordered relation with a specified number of tuples containing no indexed fields. The actual printing time will be slightly less since fewer interpreter integrity checks will be performed for PRINTREL.

The characteristics of a database are not affected by the PRINTREL command.

The PRINTREL command was slightly faster for unordered relations than ordered relations (Test 26), perhaps because unordered relations resulted in fewer system page faults. This conclusion is contradictory to what is expected, however, since unordered relations require accessing a separate B-tree relation to determine the next tuple in the unordered relation. Maybe the discrepancy is because an ordered relation with one user-defined field actually has two fields since a NEXTTID field is also present and must be printed.

NOTE: If an unordered relation has no INDEXED fields, the page fragmentation problem described in Test 26 will not occur since using each tuple's TID for ordering will result in tuples on the same page being printed after each other.

Test 28

Print an Unordered Relation, Indexing by a Field

Test: Determine the average amount of time required by the PRINTREL_FN command to print an unordered relation containing 50 tuples, each with one field. The tuples will be ordered by a specific field. This command is meaningless for ordered relations since fields cannot be indexed.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
L		10	1438.06	694.9	32.0	32.0	0.0
	256	20	1462.16	721.2	21.0	21.0	0.0
H		30	1469.52	949.8	0.2	0.4	0.0
		10	1451.30	654.0	13.0	13.0	0.0
	512	20	1447.64	714.2	0.0	0.3	0.0
		30	1452.00	658.5	0.0	0.3	0.0
		10	1459.76	664.1	8.0	8.0	0.0
	1024	20	1457.26	725.1	0.0	0.2	0.0
		30	1455.60	908.6	0.0	0.2	0.0

Analysis: There do not appear to be any major differences between the values recorded for the various page sizes. In general, 10 page buffers had the fewest system page faults, even though many more database page faults were observed. Other than the 512/30 combination, it seems apparent that more page buffers means more system page faults; this had no effect on the execution time, though.

The characteristics of a database are not affected by the PRINTREL_FN command.

The results for ordering by some indexed field is slightly better than ordering by the tuple's TID (Test 27). This is unusual since ordering by TID results in printing all tuples on one page and then moving on to the next page, while ordering by an indexed field could result in disk accesses that hop from page to page. In spite of having more database reads, ordering by an indexed field yielded fewer system page faults than Test 27.

The performance of the PRINTREL_FN command will depend on which indexed field is chosen for ordering and the distribution of tuples on pages.

Test 29

Delete a Tuple in an Unordered Relation

Test: Determine the average amount of time required to delete a tuple in an unordered relation using the DELETE_TUPLE command. The tuple contains a single non-indexed character field.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	218.28	156.7	0.6	0.6	0.0
L	256	20	216.58	135.3	0.4	0.4	0.0
		30	216.72	149.6	0.3	0.3	0.0
		10	427.70	280.1	0.2	0.1	0.0
	512	20	433.48	296.2	0.1	1.6	0.0
		30	420.76	275.6	1.6	0.0	0.0
		10	566.72	399.5	0.1	0.1	0.0
H	1024	20	579.98	426.2	0.0	0.1	0.0
		30	568.36	361.5	0.0	0.1	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-24	-8	-5
# Tuples	-107	-103	-101
# Relations	-0	-0	-0
# Prototypes	-0	-0	-0
# SDSs	-0	-0	-0

Analysis: From the execution times recorded, it is apparent that smaller pages resulted in better execution times and fewer system page faults. The number of database page faults does not seem to affect the execution time (consider 256/10 vs 1024/10). Since an unordered relation has an accessing B-tree, DELETE_TUPLE also caused the tuple's TID to be removed from the B-tree (potentially requiring the B-tree to be rebalanced). When a tuple is deleted, the next tuple in the relation will automatically be read in, again requiring access to the relation's B-tree to determine the TID of the next tuple in the unordered relation. Pages belonging to the relation will be freed when they become empty.

The number of tuples deleted includes 50 tuples for the RELCATALOG entries, plus some number of $$RELCATALOGRELNAME B-tree tuples. Note that eliminating database pages did not lower the number of blocks used by the database system. This observation will be apparent as other delete commands are issued.

Test 30

Delete a Tuple in an Ordered Relation

Test: Determine the average amount of time required to delete a tuple in an ordered relation using the DELETE_TUPLE command. The tuple has one non-indexed field, in addition to the NEXTTID field present in every tuple of an ordered relation.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	0.64	0.2	0.1	0.1	0.0
	256	20	0.74	0.2	0.1	0.1	0.0
		30	0.64	0.9	0.0	0.1	0.0
L		10	0.54	0.5	0.0	0.0	0.0
H	512	20	0.90	1.0	0.0	0.0	0.0
		30	0.60	1.1	0.0	0.0	0.0
		10	0.64	0.3	0.0	0.0	0.0
H	1024	20	0.90	0.2	0.0	0.0	0.0
		30	0.72	1.2	0.0	0.0	0.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-7	-6	-2
# Tuples	-50	-50	-50
# Relations	-0	-0	-0
# Prototypes	-0	-0	-0
# SDSs	-0	-0	0

Analysis: Based on the results obtained, all combinations appear fairly equal. For all page sizes, the combinations with 20 page buffers were slower. This is likely a fluke and can be considered insignificant since the execution times are so small. When a tuple is deleted, the next tuple in the relation will be automatically read in. Pages belonging to the relation will be freed up when they become empty.

Compared with the Test 29 results (DELETE_TUPLE for unordered relations), one can see the difference having to access and update and rebalance the accessing B-tree for an unordered relation can cause. Deleting a tuple in an unordered relation is much slower.

Test 31

Delete an Unordered Relation

Test: Determine the average amount of time required to delete an unordered relation using the DELETEREL command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	564.86	297.1	20.8	20.8	11.1
	256	20	560.48	327.2	7.2	7.2	3.6
L		30	551.58	337.2	5.1	5.4	2.7
		10	786.38	456.1	7.2	7.2	4.4
	512	20	776.52	473.0	2.6	2.8	1.2
		30	776.42	470.2	2.1	2.5	0.8
		10	1099.98	754.0	3.5	3.5	0.4
H	1024	20	1102.96	713.4	1.6	1.8	0.5
		30	1084.94	656.2	1.3	1.8	0.3

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-187	-97	-71
# Tuples	-284	-268	-257
# Relations	-50	-50	-50
# Prototypes	-0	-0	-0
# SDSs	-0	-0	-0

Analysis: The execution time appears to be tied to the page size; larger pages resulted in slower execution times and more

system page faults. The fact that the increased number of database page faults caused by smaller pages did not result in slower execution times is quite unusual (and unexpected).

Deleting a relation involves 1) accessing the Relational Catalog information for the relation and its prototype, 2) deleting every page used by the relation, 3) deleting the relation's entry in the RELCATALOG and $$RELCATALOGRELNAME relations. For an unordered relation, the same steps must also be performed for the B-trees for each indexed field, plus the relation's accessing B-tree. Since the relations being deleted have similar names, the B-tree for the Relational Catalog was rebalanced frequently.

The reduction in the number of tuples includes 50 RELCATALOG tuples for the 50 relations deleted, plus however many tuples were recovered from the RELCATALOG B-tree as relations were deleted.

Test 32

Delete an Ordered Relation

Test: Determine the average amount of time required to delete an ordered relation using the DELETEREL command.

Results:

Performance Measurement Statistics - For One Command

Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
	10	483.36	74.6	11.3	11.3	5.5
256	20	477.46	82.1	3.9	4.0	1.4
L	30	473.30	69.3	3.5	3.8	1.2
	10	592.34	54.4	3.6	3.6	1.2
512	20	598.08	81.3	1.9	2.1	0.5
	30	592.76	246.8	1.5	1.8	0.4
	10	699.28	153.8	1.5	1.6	0.3
1024	20	699.52	174.6	1.2	1.5	0.2
H	30	702.72	173.0	1.0	1.4	0.2

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-71	-18	-11
# Tuples	-116	-106	-103
# Relations	-50	-50	-50
# Prototypes	-0	-0	-0
# SDSs	-0	-0	-0

Analysis: Smaller page sizes resulted in better execution times. The relationship between the execution time and the page size

is not quite as obvious (compare 512/10 and 256/20). Prior to the deletion, the database had to be accessed to read in the relation's prototype. Database reads and writes were needed to sequentially access and free every page attached to the relation.

Deleting a relation involves 1) accessing the Relational Catalog information for the relation and its prototype, 2) deleting every page used by the relation 3) deleting the relation's entry in the RELCATALOG and $$RELCATALOGRELNAME relations. Since the relations being deleted have similar names, the B-tree for the Relational Catalog was rebalanced frequently.

When compared to the results for unordered relations (Test 31), one can see the improvement in execution time produced by not having to access and delete pages associated with a B-tree for an unordered relation. Not having an accessing B-tree improved every measurement statistic recorded.

Test 33

Delete a Prototype

Test: Determine the average amount of time required to delete a relational prototype using the DELETE_PROTO command. Deleting an SDS prototype should take the same amount of time.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	132.74	46.9	3.6	3.6	2.3
	256	20	131.46	86.6	2.5	2.6	1.4
		30	128.46	48.4	2.3	2.6	1.3
		10	132.94	95.5	1.8	1.8	0.7
L	512	20	127.82	57.0	1.3	1.6	0.4
		30	129.12	71.1	1.1	1.5	0.3
H		10	190.66	143.6	1.3	1.4	0.3
	1024	20	186.10	117.8	1.0	1.2	0.1
		30	187.72	143.1	0.8	1.2	0.1

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-74	-70	-13
# Tuples	-127	-107	-103
# Relations	-50	-50	-50
# Prototypes	-50	-50	-50
# SDSs	-0	-0	-0

Analysis: The results from the 256 and 512-byte tests appear similar, in spite of the variations in the number of database and system page faults. Database reads and writes were required to sequentially access and free every page attached to the prototype.

Deleting a prototype involves 1) accessing the Relational Catalog information for the prototype, 2) deleting every page used by the prototype, and 3) deleting the prototype's entry in the RELCATALOG and $$RELCATALOGRELNAME relations. Since the names of the prototypes used were similar, the B-tree for the Relational Catalog was rebalanced frequently.

The tuples and pages recovered include those used by the prototype, plus tuples from the RELCATALOG and $$RELCATALOGRELNAME relations. Since each relational prototype is implemented as a relation, 50 relations were also recovered.

Compared to Test 32 (Deleting an Ordered Relation), having no relational prototype tc also access improves the execution time, although the prototypes involved in this test had fewer tuples than the relations in Test 32.

Test 34

Create an SDS Prototype

Test: Determine the average amount of time required to create a prototype for a spatial data structure using the CREATE_SDSPROTO command. The prototype will identify the prototypes of relations that will comprise this type of spatial data structure.

Results:

Performance Measurement Statistics - For One Command

Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
	10	322.86	184.9	2.0	0.5	1.7
256	20	327.64	163.3	1.4	0.1	1.2
	30	328.70	202.8	1.2	0.1	1.0
	10	302.40	163.6	0.5	0.1	0.4
512	20	322.19	180.1	0.2	0.1	0.1
	30	358.02	203.7	0.1	0.1	0.0
	10	353.30	254.0	0.2	0.1	1.4
1024	20	355.20	204.1	0.0	0.1	0.0
	30	354.96	180.2	0.0	0.1	0.0

L

H

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+36	+23	+20
# Pages used	+71	+23	+10
# Tuples	+116	+109	+103
# Relations	+50	+50	+50
# Prototypes	+50	+50	+50
# SDSs	+0	+0	+0

Analysis: There do not appear to be any relationships between the page size and execution time, especially since the best and worst execution times both involve the same page size.

Creating an SDS prototype is similar to the creation of a relational prototype; A RELCATALOG and $$RELCATALOGRELNAME entry must be created for the prototype.

Since an SDS prototype is implemented as an ordered relation, one tuple is required for the NEXTTID tuple in each prototype. The other new tuples are for RELCATALOG and $$RELCATALOGRELNAME tuples.

Test 35

Add a Relational Prototype to an SDS Prototype

Test: Determine the average amount of time required to add the name of a relational prototype to an SDS prototype using the ADDTO_SDSPROTO command. The type of relation defined by the relational prototype (unordered or ordered) is unimportant since the relational prototype will not be accessed until the SDS prototype is used to create an SDS.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	119.50	68.2	2.8	1.8	1.9
H	256	20	119.84	79.8	2.8	1.9	1.8
		30	117.14	33.0	2.5	1.9	1.6
		10	115.58	71.2	1.6	0.7	1.2
L	512	20	114.18	88.0	1.5	0.7	1.0
		30	114.94	58.0	1.3	0.7	0.9
		10	116.26	76.4	1.3	0.3	1.0
	1024	20	116.16	85.6	1.1	0.3	0.9
		30	115.92	67.0	0.9	0.3	0.7

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+25	+51	+100
# Pages used	+50	+50	+50
# Tuples	+50	+50	+50
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: There do not appear to be any relationships between the page size, the number of system page faults, and the execution time, although the 512-byte pages performed slightly better as a group.

Adding a relational prototype to an SDS prototype is similar to adding a field to a relational prototype. For SDS prototypes, however, the database system must also verify the existence of the relational prototype by searching the $$RELCATALOGRELNAME B-tree for the name of the prototype.

One tuple occupying one page was added to <u>each</u> SDS prototype. This tuple contains the name of a relational prototype to be associated with the SDS prototype.

Test 36

Creating a SDS with One Unordered Relation

Test: Determine the average amount of time required to create a spatial data structure using the CREATE_SDS command. The SDS prototype used will cause an unordered relation to automatically be created and attached to the spatial data structure. The unordered relation will be empty. An accessing B-tree will also be created for the unordered relation.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	2004.40	955.0	48.5	42.4	17.7
	256	20	1956.98	957.4	22.7	17.6	8.9
L		30	1688.02	847.0	10.0	5.9	5.5
		10	2110.32	1014.7	24.3	21.3	8.4
	512	20	2058.08	991.4	8.1	6.0	3.7
		30	2017.74	968.8	5.1	3.1	2.3
		10	2378.32	1327.2	14.6	13.4	5.4
	1024	20	2369.70	1260.3	4.1	2.7	1.8
H		30	2377.64	1235.3	3.8	2.6	1.6

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+135	+117	+167
# Pages used	+264	+116	+83
# Tuples	+400	+372	+361
# Relations	+150	+150	+150
# Prototypes	+0	+0	+0
# SDSs	+50	+50	+50

Analysis: From the results, it appears that decreasing the page size lowers both the number of system page faults and the execution time.

The process of creating a SDS involves lots of database accesses. The sequence is as follows: 1) the SDS prototype is accessed, 2) RELCATALOG and $$RELCATALOGRELNAME entries are created for the SDS, 3) for each tuple in the SDS prototype, a CREATEREL command is to create a uniquely named relation (and accessing B-tree) using the prototype identified in the tuple, 4) the relation is attached to the SDS, and 5) the SDS prototype is released. The entire process involves quite a bit of searching for specific Relational Catalog tuples, and rebalancing of the $$RELCATALOGRELNAME B-tree.

The number of relations created includes 50 SDSs, plus one unordered relation and accessing B-tree for each SDS. The increase in the number of tuples includes tuples for the RELCATALOG and $$RELCATALOGRELNAME relations, plus one tuple for each SDS relation to hold the unique name of the relation attached to the SDS.

Test 37

Creating a SDS with One Ordered Relation

Test: Determine the average amount of time required to create a spatial data structure using the CREATE_SDS command. The SDS prototype used will cause an ordered relation to automatically be created and attached to the spatial data structure. The ordered relation will be empty.

Results:

Performance Measurement Statistics - For One Command

Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
	10	1754.98	271.9	36.6	32.7	13.0
256	20	1756.02	268.3	17.6	13.8	6.0
	30	1749.34	264.3	9.0	5.4	4.1
	10	1730.20	283.0	16.5	14.7	5.7
512	20	1694.72	278.5	5.2	3.6	2.0
	30	1716.16	294.7	4.6	3.2	1.9
	10	1854.80	305.6	11.7	10.9	3.7
L 1024	20	1622.14	328.6	3.8	2.6	1.5
H	30	1868.92	320.0	3.6	2.7	1.4

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+99	+192	+145
# Pages used	+194	+92	+72
# Tuples	+285	+263	+257
# Relations	+100	+100	+100
# Prototypes	+0	+0	+0
# SDSs	+50	+50	+50

Analysis: Given the statistics recorded, it is hard to determine the relationship between the execution time, the number of page buffers, the page size, and the number of system page faults. In general, having more database page faults did not affect the execution time (compare 256/10 to 256/20).

The process of creating a SDS involves lots of database accesses. The sequence is as follows: 1) the SDS prototype is accessed, 2) RELCATALOG and $$RELCATALOGRELNAME entries are created for the SDS, 3) for each tuple in the SDS prototype, a CREATEREL command is to create a uniquely named relation using the prototype identified in the tuple, 4) the relation is attached to the SDS, and 5) the SDS prototype is released. The entire process involves quite a bit of searching for specific Relational Catalog tuples, and rebalancing of the $$RELCATALOGRELNAME B-tree.

The number of relations created includes 50 SDSs, plus one ordered relation for each SDS. The increase in the number of tuples includes tuples for the RELCATALOG and $$RELCATALOGRELNAME relations, plus one tuple for each SDS relation to hold the unique name of the relation attached to the SDS.

Creating an SDS with an ordered relation is faster than its unordered relation counterpart (Test 36) since ordered relations have no accessing B-trees. Having no B-tree caused every measured statistic to be smaller.

Test 38

Printing a SDS with One Unordered Relation

Test: Determine the average amount of time required by the PRINTSDS command to print a spatial data structure consisting of a single unordered relation with 50 tuples. Since the relation is unordered, its accessing B-tree will be used to determine the order in which tuples should be printed.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	1572.20	980.0	26.0	26.0	0.0
	256	20	1568.22	993.9	24.9	24.9	0.0
		30	1574.70	1007.1	0.2	0.4	0.0
		10	1571.30	944.5	16.0	16.0	0.0
	512	20	1581.54	1010.2	0.1	0.3	0.0
		30	1585.28	1060.7	0.0	0.3	0.0
H		10	1597.44	1035.6	9.0	9.0	0.0
L	1024	20	1549.46	965.6	0.0	0.2	0.0
		30	1588.66	993.7	0.0	0.0	0.0

Analysis: The trend seems to be that more system page faults caused slower execution times. The number of system page faults does not appear related to the page size, though the same can be said about the relationship between the number of database page faults (which was high for the smaller 256-byte pages), and the execution time.

Processing for the PRINTSDS command includes commands that have already been tested: ACCESSREL, PRINTREL for each relation attached to the SDS, NEXT_TUPLE to get to the next tuple in the SDS, and RELEASEREL. Given that knowledge, it is no major challenge to predict the time required to print an SDS with some specified number of relations. The actual printing time will be slightly less since fewer interpreter integrity checks will be performed for PRINTSDS.

The characteristics of a database are not changed by the PRINTSDS command.

When compared to Test 27 (PRINTREL for a single unordered relation), it appears that printing that same relation from within an SDS adds approximately 100 ms to the execution time, and requires 2-3 more database page faults.

Test 39

Printing a SDS with One Ordered Relation

Test: Determine the average amount of time required by the PRINTSDS command to print an SDS that consists of a single ordered relation with 50 tuples. The NEXTTID field in each tuple of the ordered relation will be used to determine the order in which tuples should be printed.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	1586.76	352.7	21.0	21.0	0.0
	256	20	1585.72	263.0	16.0	16.0	0.0
		30	1575.32	299.6	0.1	0.3	0.0
		10	1592.88	309.9	11.0	11.0	0.0
	512	20	1584.80	284.6	0.0	0.2	0.0
		30	1570.06	373.0	0.0	0.2	0.0
L		10	1554.26	293.3	7.0	7.0	0.0
	1024	20	1584.94	322.6	0.0	0.2	0.0
H		30	1596.90	266.9	0.0	0.2	0.0

Analysis: There are no major relationships between the execution times and the number of system page faults. As expected, fewer page buffers resulted in more reads, although this did not affect the overall execution time.

Processing for the PRINTSDS command includes commands that have already been tested: ACCESSREL, PRINTREL for each relation attached to the SDS, NEXT_TUPLE to get to the next

tuple in the SDS, and RELEASEREL. Given that knowledge, it is no major challenge to predict the time required to print an SDS with some specified number of relations. The actual printing time will be slightly less since fewer interpreter integrity checks will be performed for PRINTSDS.

The characteristics of a database are not changed by the PRINTSDS command.

The results obtained from this test are similar to those from Test 38 (PRINTSDS with an unordered relations).

When compared with Test 32 (PRINTREL for a single ordered relation), one can see that including an ordered relation within an SDS adds approximately 60 ms and 1-4 more database page faults to the measured statistics. Similar increases were observed in Test 38 for unordered relations within SDSs.

Test 40

Attaching a Relation to an SDS

Test: Determine the average amount of time required to attach a relation to a spatial data structure using the ATTACHREL command.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	439.10	213.4	19.5	19.5	3.1
	256	20	432.92	209.6	8.2	8.2	2.1
		30	429.78	204.4	2.9	2.9	1.8
		10	437.58	190.1	15.4	15.4	2.1
	512	20	428.72	219.5	2.1	2.1	1.6
		30	428.38	196.1	1.9	2.2	1.7
		10	434.12	216.0	9.2	9.2	1.5
L	1024	20	420.60	230.9	1.5	1.6	1.3
		30	428.24	207.2	1.3	1.6	1.0

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	+1	+0	+0
# Pages used	+2	+0	+0
# Tuples	+50	+50	+50
# Relations	+0	+0	+0
# Prototypes	+0	+0	+0
# SDSs	+0	+0	+0

Analysis: There are no major differences between the various combinations tested. The execution times and number of system page faults are fairly consistent, in spite of the differences in the number of database page faults.

Attaching a relation to an SDS involves accessing the SDS and its prototype, verifying that the relation to be attached does exist, adding one tuple to the SDS relation containing the name of the relation, and releasing both the SDS and its prototype. Note that the relation to be attached does not have to be accessed.

One tuple was created for each relation added to the SDS. Since the SDS already had relations attached to it (those automatically attached by the CREATE_SDS command), the new tuple was added to pages already assigned to the SDS.

Test 41

Removing a Relation from an SDS

Test: Determine the average amount of time required to remove a relation from a spatial data structure using the REMOVEREL command

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
H		10	169.08	68.3	6.6	6.6	2.0
L	256	20	158.90	31.3	3.0	3.0	1.8
		30	168.82	78.2	2.8	3.1	1.6
		10	163.02	31.2	2.2	2.2	1.7
	512	20	162.64	32.1	2.0	2.2	1.5
		30	161.62	22.9	1.8	2.2	1.3
		10	161.48	20.8	1.6	1.6	1.2
	1024	20	160.94	25.0	1.4	1.4	1.2
		30	165.94	26.9	1.2	1.2	0.9

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-2	-0	-50
# Tuples	-50	-50	-50
# Relations	-0	-0	-0
# Prototypes	-0	-0	-0
# SDSs	-0	-0	-0

Analysis: In spite of varying page sizes and number of system page faults, the execution times observed for the various combinations did not differ significantly. As expected, more and bigger page buffers reduced the database-recorded statistics. No combinations resulted in no reads since 50 relations could not fit into 30 page buffers.

Removing a relation from an SDS involves accessing the SDS and its prototype, verifying that the relation to be removed does exist, searching the SDS relation for the tuple containing the name of the relation, deleting that one tuple, and releasing both the SDS and its prototype. Note that the relation to be removed does not have to be accessed.

One tuple was deleted for each relation removed from the SDS. Since the SDS still has other relations attached to it, deleting the tuple did not require freeing any pages assigned to the SDS.

Test 42

Deleting a SDS with One Unordered Relation

Test: Determine the average amount of time required by the DELETE_SDS command to delete an SDS consisting of a single unordered relation with 50 tuples.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	632.74	67.0	35.6	35.6	19.1
	256	20	619.16	47.4	18.3	18.4	11.2
L		30	553.18	66.2	9.0	9.3	5.3
		10	1035.04	210.9	19.8	19.8	11.1
	512	20	984.78	205.9	16.7	6.8	3.0
		30	1005.40	203.4	4.0	4.4	1.3
		10	1827.88	426.3	10.9	11.0	6.4
H	1024	20	1859.06	450.8	3.3	3.5	1.1
		30	1839.28	462.9	2.7	3.1	0.7

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-282	-122	-37
# Tuples	-508	-472	-461
# Relations	-150	-150	-150
# Prototypes	-0	-0	-0
# SDSs	-50	-50	-50

Analysis: In general, larger pages resulted in more database and system page faults and slower execution times.

Deleting the SDS involved many database accesses. First, the SDS and its prototype had to be accessed. Next, each tuple in the SDS relation was read. For each relation mentioned in a tuple, a DELETEREL command (with all of its overhead) was issued if the relation was not attached to the SDS via the ATTACHREL command. Then, the RELCATALOG and $$RELCATALOGRELNAME entries for the SDS were deleted. Finally, the SDS prototype was released. Note that the DELETEREL command will be concerned with deleting any B-trees associated with the unordered relation.

One tuple was recovered for each SDS, plus 100 RELCATALOG tuples (50 each for the unordered relation and 50 for its accessing B-tree), plus however many $$RELCATALOGRELNAME tuples were needed to store the additional 100 B-tree keys.

Deleting a spatial data structure with a single unordered relation took approximately 800 ms longer than deleting a single unordered relation (Test 31).

Test 43

Deleting a SDS with One Ordered Relation

Test: Determine the average amount of time required by the DELETE_SDS command to delete a spatial data structure consisting of a single ordered relation with 50 tuples.

Results:

Performance Measurement Statistics - For One Command

	Page Size	# Buffers	Average Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	612.60	67.9	27.1	27.1	13.0
	256	20	598.60	63.5	10.9	11.0	4.1
L		30	596.64	65.7	6.5	6.8	2.9
		10	721.08	130.8	12.7	12.7	4.8
	512	20	714.14	121.9	3.4	3.6	0.9
		30	712.42	126.1	3.1	3.5	0.8
H		10	1429.26	348.4	9.1	9.2	2.6
	1024	20	1401.76	324.0	2.9	3.2	0.7
		30	1361.30	310.1	2.4	2.9	0.4

Changes to Database Characteristics - For 50 Commands

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	-0	-0	-0
# Pages used	-198	-105	-76
# Tuples	-332	-313	-307
# Relations	-100	-100	-100
# Prototypes	-0	-0	-0
# SDSs	-50	-50	-50

Analysis: In general, larger pages resulted in more database and system page faults, and slower execution times.

Deleting the SDS involved many database accesses. First, the SDS and its prototype had to be accessed. Next, each tuple in the SDS relation was read. For each relation mentioned in a tuple, a DELETEREL command (with all of its overhead) was issued if the relation was not attached to the SDS via the ATTACHREL command. Then, the RELCATALOG and $$RELCATALOGRELNAME entries for the SDS were deleted. Finally, the SDS prototype was released.

One tuple was recovered for each SDS, plus 50 RELCATALOG tuples and however many $$RELCATALOGRELNAME tuples were needed to store the additional 50 B-tree keys.

Deleting a spatial data structure with a single ordered relation took approximately 500 ms longer than deleting a single ordered relation (Test 32). As expected, the execution times for ordered relations were better than those for unordered relations (Test 42) since no accessing B-trees were involved.

Test 44

Creating the Sample Database

Test: Determine the average amount of time required to create the sample database described in [49]. Unlike the earlier tests, this test was provided mainly to show that the database commands can be used to perform useful functions. The sample database was created once, not 50 times like in other tests.

Results:

Performance Measurement Statistics - For Entire Sequence

	Page Size	# Buffers	Execution Time (ms)	# System Recorded Page Faults	# Database Recorded Page Faults	# Reads	# Writes
		10	19104.00	2568.0	93.0	29.0	88.0
	256	20	18916.00	2855.0	57.0	3.0	57.0
L		30	18799.00	2677.0	45.0	1.0	45.0
		10	19406.00	3035.0	31.0	1.0	31.0
	512	20	19204.00	2274.0	21.0	1.0	21.0
		30	19281.00	2724.0	11.0	1.0	11.0
		10	21525.00	2669.0	24.0	1.0	24.0
	1024	20	21321.00	3623.0	14.0	1.0	14.0
H		30	21604.00	2869.0	4.0	1.0	4.0

Database Characteristics

Page Size	256 bytes	512 bytes	1024 bytes
# VMS Blocks	46	44	73
# Pages used	77	42	35
# Tuples	124	118	117
# Relations	28	28	28
# Prototypes	25	25	25
# SDSs	0	0	0

Analysis: In general, smaller pages yielded better execution times. This may change when actual relations and spatial data structures are created. The number of database page faults did not factor into the execution time (compare the 256/10 and 1024/10 results). Although there were lots of writes to disk, few reads were recorded since creating a new relation, SDS, or prototype is simply a write operation. More reads would be expected if any of the prototypes were used to create actual relations or spatial data structures.

Of the 28 relations created, 4 were for system relations, 16 were for relational prototypes, and 8 were for SDS prototypes.

Summary-Highest/Lowest Execution Times

The following chart summarizes the number of times a particular page size/# buffers combination had the highest or lowest execution time. This chart will be used to identify trends, but cannot be used to identify the best combination since the chart does not show whether the difference between the high and low value was significant.

Page Size	# Buffers	Highest Time	Lowest Time
	10	8	1
256	20	3	7
	30	4	10
	10	0	10
512	20	4	5
	30	3	2
	10	4	5
1024	20	6	4
	30	15	3

From the chart, it is apparent that a page size of 1024 bytes is not practical. Of the 1024-byte combinations tested, only 1024/10 had more Lowest times than Highest, and that difference was not by much. The other 1024-byte combinations did worse.

The 512-byte combinations performed better than their 256-byte counterparts, perhaps because 512 bytes is the page size used by VMS. When a 256-byte page was required by the database system, a 512-byte VMS block was still read in, with even chances that a later database request will not use the other 256-byte page in the block

Summary-Ordered Relation vs Unordered Relation

Table 10.1 on the following page contrasts the execution times for unordered and ordered relations, for all page size/# buffer combinations.

Entries in the table are rounded to the nearest tenths. This is not only to fit the entire width of the table on a page, but because the chart is to identify trends only; more accurate values can be found with the individual tests.

Other than Tests 5 and 6 (CREATE_RELPROTO), ordered relations always performed better than their unordered counterparts. In some cases (Tests 29 and 30), the differences between the statistics recorded were significant. The importance of these differences depends on how often the respective commands are issued. For example, the CREATE_RELPROTO command (Tests 5 and 6), is not executed as often as the BUILD_TUPLE (Tests 13 and 14) and DELETE_TUPLE (Tests 29 and 30) commands. As such, the fact that Tests 5 and 6 indicated that CREATE_RELPROTO was slower for ordered relations is not that important.

Also, comparing the results from ordered relations and unordered relations was slightly unfair. Most of the tests performed would naturally favor ordered relations since the bulk of the database operations performed involved accessing all tuples in a relation chronologically. A fairer test would be to compare the PRINTREL_FN command (print an unordered relation using a specific field) to whatever would be required to perform the same tuple ordering with an ordered relation. Searches for tuples with specific field values would also be fair tests.

Table 10.1 - Execution Times for Unordered vs Ordered Operations

Page Size	256			512			1024		
# Buffers	10	20	30	10	20	30	10	20	30
Test 5 - U	288.3	292.1	293.1	211.0	318.7	330.4	489.2	491.1	513.2
6 - O	502.3	502.6	499.1	533.5	534.0	534.8	708.9	711.8	712.6
Test 7 - U	205.3	204.2	204.5	127.9	197.3	197.2	197.1	197.4	204.3
8 - O	207.8	205.9	207.8	204.0	209.8	207.1	207.7	207.3	205.1
Test 10 - U	994.7	931.8	934.1	1037.9	1030.5	1029.2	1284.3	1283.8	1286.0
11 - O	733.8	727.7	727.2	712.4	707.9	708.3	760.0	759.6	762.7
Test 13 - U	133.6	132.7	133.4	128.6	127.3	127.3	110.1	112.5	111.8
14 - O	46.0	44.9	46.8	45.4	46.0	45.9	45.3	46.2	45.7
Test 15 - U	0.8	0.8	0.8	0.7	0.8	0.8	0.7	0.7	0.6
16 - O	0.8	0.7	0.6	0.6	0.5	0.6	0.6	0.5	0.7
Test 20 - U	23.3	22.9	22.7	22.8	23.1	23.4	22.8	22.6	22.5
21 - O	22.9	23.2	23.0	22.8	23.1	23.1	22.8	22.8	22.9
Test 22 - U	394.4	383.9	386.7	389.5	382.8	383.0	391.0	387.6	387.6
23 - O	428.6	428.3	428.6	428.7	428.4	428.4	428.8	428.6	428.3
Test 22 - U	24.3	23.8	23.4	23.6	24.8	23.5	23.3	23.5	23.5
23 - O	23.8	23.5	23.6	23.6	23.6	23.5	23.4	23.6	23.6
Test 24 - U	23.8	23.5	23.2	23.9	23.7	24.8	23.4	23.4	23.6
25 - O	23.3	22.9	27.0	23.3	23.5	23.7	23.2	23.5	23.1
Test 26 - U	1521.1	1508.3	1524.3	1520.9	1511.7	1527.8	1524.0	1549.8	1508.7
27 - O	1464.7	1468.2	1470.4	1479.2	1483.8	1464.0	1463.2	1489.7	1471.1
Test 29 - U	218.3	216.6	216.7	427.7	433.5	420.8	566.7	580.0	568.4
30 - O	0.6	0.7	0.6	0.5	0.9	0.6	0.6	0.9	0.7
Test 31 - U	564.9	560.5	551.6	786.4	776.5	776.4	1100.0	1103.0	1084.9
32 - O	483.4	477.5	473.3	592.3	598.1	592.8	699.3	699.5	702.7
Test 36 - U	2004.4	1957.0	1688.0	2110.3	2058.1	2017.7	2378.3	2369.7	2377.6
37 - O	1755.0	1756.0	1749.3	1730.2	1694.7	1716.2	1854.8	1622.1	1868.9
Test 38 - U	1586.8	1585.7	1575.3	1592.9	1584.8	1570.1	1554.3	1584.9	1596.9
39 - O	1572.2	1568.2	1574.0	1571.3	1581.5	1585.3	1597.4	1549.5	1588.7
Test 42 - U	632.7	619.2	553.2	1035.0	984.8	1005.4	1827.9	1859.1	1839.3
43 - O	612.6	598.6	596.6	721.1	714.1	712.4	1429.3	1401.8	1361.3

Space Utilization

Table 10.2 on the following page illustrates how much VMS and database space was required by each page size/# buffers combination. This data was summarized from the individual statistics for each test and represents the accumulated totals from 50 commands, not one. Positive numbers identify commands that create, while negative numbers are produced by commands that delete.

General Observations -

1. As has been observed earlier, the number of VMS blocks is directly related to the number of pages used.
   - With 256-byte pages, the number of blocks is almost half of the number of pages. This implies that VMS was generally able to pack two 256-byte page into one of its 512-byte blocks. The existence of a few extra blocks indicates that there are cases where a 256-byte database page was assigned an entire block, with half of the block being unused.
   - With 512-byte pages, the ratio almost 1:1, implying that one 512-byte VMS block was allocated for each 512-byte database page.
   - With 1024-byte pages, one can see that one page generally occupied two VMS blocks. As with the 256-byte pages, some of
2. When keys are added or deleted in a B-tree (e.g. an unordered relation's accessing B-tree), the change in the number of tuples does not always match what is expected. This is because the number of keys that can fit in each B-tree node depends on the page size and key length. For example, if five keys of a particular length can fit in a B-tree node, the addition of six tuples to an unordered relation may result in only two B-tree tuples being added to the relation's accessing B-tree.

Table 10.2 - Space Utilization for Unordered vs Ordered Operations

Page Size		256			512			1024		
		# Blocks	# Pages	# Tuples	# Blocks	# Pages	# Tuples	# Blocks	# Pages	# Tuples
CREATE_RELPROTO										
Test	5 - U	+36	+71	+117	+22	+21	+107	+20	+9	+103
	6 - O	+61	+121	+167	+72	+71	+157	+120	+59	+153
ADDTO_RELPROTO										
Test	7 - U	+25	+50	+50	+50	+50	+50	+100	+50	+50
	8 - O	+0	+0	+50	+0	+0	+50	+0	+0	+50
CREATEREL										
Test	10 - U	+72	+142	+233	+46	+46	+217	+44	+22	+207
	11 - O	+36	+71	+116	+23	+23	+109	+70	+10	+103
BUILD_TUPLE										
Test	13 - U	+10	+19	+108	+10	+9	+104	+8	+4	+101
	14 - O	+4	+7	+50	+3	+3	+50	+4	+2	+50
DELETE_TUPLE										
Test	29 - U	-0	-24	-107	-0	-8	-103	-0	-5	-101
	30 - O	-0	-7	-50	-0	-6	-50	-0	-2	-50
DELETEREL										
Test	31 - U	-0	-187	-284	-0	-97	-268	-0	-71	-257
	32 - O	-0	-71	-116	-0	-18	-108	-0	-11	-103
CREATE_SDS										
Test	36 - U	+135	+264	+400	+117	+116	+372	+167	+83	+361
	37 - O	+99	+194	+285	+192	+92	+263	+145	+72	+257
DELETE_SDS										
Test	42 - U	-0	-282	-508	-0	-122	-472	-0	-37	-461
	43 - O	-0	-198	-332	-0	-105	-313	-0	-76	-307

3. Since a B-tree must always remain balanced, adding some number of keys and then deleting those same keys occassionally resulted in different final B-trees, depending on how the keys were distributed within individual B-tree nodes.
4. All tests that involve deleting showed an interesting feature. For each of those tests, a reduction in the number of pages did not change the number of blocks used by the database system. This implies that once VMS has allocated those blocks to the database file, they are ours to keep (and potentially reuse). This phenomenon is not desirable since it it implies that a database will always require more VMS blocks, even if the database itself shrinks in size.

Test 5/6 - This pair is unique because this is the only case where unordered relations used less space than their ordered counterparts. This is because a prototype for an unordered relation is empty, while a similar prototype for an ordered relation has one tuple (NEXTTID) occupying one page for each of the 50 prototypes. Since an empty relational prototype is worthless, the differences in space utilization at this point are insignificant.

Test 7/8 - These results are more representative than those from Tests 5 and 6. Each relational prototype now has one page assigned for tuples. For ordered relations, there are two tuples on the page, while for unordered relations, there is only one. Note that adding the number of blocks from Test 5 to those from Test 7 yields the same space utilization as in Test 6. The same can be said for the number of pages.

Test 11/12 - Note that with this comparison, ordered relations perform much better in terms of space utilization. This is because unordered relations require a separate accessing B-tree which must have a tuple in the RELCATALOG and $$RELCATALOGRELNAME relations. Ordered relations use the NEXTTID field within each tuple and have no accessing B-tree.

Test 13/14 - These results are similar to those for Tests 11 and 12. Each tuple added to an unordered relation generates a key that must be added to the relation's accessing B-tree. No keys were needed for ordered relations.

Test 29/30 - As expected, all of the tuples created in Tests 13 and 14 must be deleted. When the number of pages and tuples don't match exactly, the deletion of tuples cause the B-trees to be balanced differently.

Test 31/32 - For each unordered relation deleted, its accessing B-tree also had to be deleted. As with Tests 29 and 30, differences from Tests 10 and 11 are likely because the $$RELCATALOGRELNAME B-tree was balanced differently after the 100 (50 for ordered relations) relation names were deleted.

Test 36/37 - As with Tests 10 and 11, an SDS with an unordered relation required extra blocks, pages, and tuples for its accessing B-tree.

Test 42/43 - For each SDS with an ordered relation, tuples were deleted in the $$RELCATALOGRELNAME and RELCATALOG relations, in addition to one tuple within the SDS relation to contain the name of the ordered relation. For SDSs with unordered relations, two additional tuples were deleted for the relation's accessing B-tree. Again, differences in the number of tuples reflect the rebalancing of the $$RELCATALOGRELNAME B-tree.

Comparison to Memory-Resident System Results

As was indicated in "Database Manipulation Commands" on page 125, the database functions provided by this project differ greatly from those documented in [49]. [49], for example, did not implement the concept of a prototype, and required the database user to be intimately familiar with the underlying storage structure of the database system. Few of the database commands provided by this project performed the same function as in [48-49], and when some similarities did exist, the disk-resident system always provided more (e.g. labeling fields when a tuple was printed).

Given these differences, it is futile to compare the performance results of this project to those for the memory-resident system, even though this project started out with that goal. The two systems are as different (both outwardly and inwardly) as apples and oranges.

Test Conclusions

For many of the tests run, there were no observed relationships between the various measurement statistics. Also, no combination was identified as "best" so no conclusions could be drawn as to what is the optimum page size and number of page buffers. To hopefully gather more conclusive data, the entire test sequence should be rerun using 200 relations instead of 50. With this additional information, trends might be identified and explanations may become apparent as to why one page size/# buffer combination performed better than another.

The number of system page faults seemed high, implying that the database system did not entirely eliminate excessive paging by VMS. This would likely be reduced if the database page buffers were locked within the working set and in memory, as would be done in a production environment.

For many tests, using different page sizes did not appear to affect the overall execution time. This may be due to the way VMS blocks reads from disk. Reading a single 256-byte page probably have resulted in VMS reading in an entire 512-byte record, but only providing the database system one of the smaller pages. This blocking on VMS's part could eliminate half of the database disk accesses, depending on whether a subsequent database request was for the other 256-byte page on the 512-byte block. Similar conclusions could be drawn about 1024-byte pages, although this time it is the database system that performed the blocking. Each read of a 1024-byte page would result in two 512-byte blocks being read, presumably by an efficient channel program. If subsequent database requests were for information on the same page (as would be likely with larger page sizes), no disk read would be necessary. System R compromises by having three buffer pools with small pages (4K), and one pool to read (with one I/O operation) much larger pages (32K).

When "delete" commands were issued, the number of blocks occupied by the database was not reduced. This will become a greater problem as a database grows in size.

Better tools need to be developed to identify <u>how</u> the database system spent its time during the execution of interpreter commands. It is suspected that a large amount of the time was spent 1) converting characters to integers and vice versa, and 2) reading pages from disk. Choosing a different programming language could eliminate number 1 and interfacing with the operating system read and write modules directly

could improve number 2. Of course, with better tools, we may find that the real "problem areas" were neither 1 or 2.

Tools also need to be developed to identify the distribution of keys on B-tree pages. By increasing or decreasing the number of keys per page, the horrible results obtained in Test 19 might be improved.

Finally, note that the names of relations, fields, and field values were chosen to encourage a worst case test. For example, in Test 19, the field values were selected to encourage tree rebalancing when the old value was deleted and when the new value was added. The results from Test 19 (and other tests) would be better if "typical values" (whatever that may mean) were used.

CHAPTER 11 - CONCLUSIONS AND FUTURE WORK

Conclusions

This project proved to be a valuable learning experience. With 25000 lines of code and comments, the database system was much larger than was originally anticipated and the lack of defined interfaces between the physical and logical structures of the system manifested itself early. Were I to rewrite the database system, I would do as the System R programmers did and start again from scratch using the knowledge obtained from this first attempt at designing a database system. I would also take some database courses to explore alternative designs, problems, and solutions. Most of my database knowledge came from the literature search and self-education and I often made mistakes that had been made many times before.

Much research has been done in the direction of Data Design Languages and schemata since this database system was first conceived. Relational and SDS prototypes are a start, but fall short of being a set of schemata that formally define how the database system operates. Any subsequent rewrite of the database system should definitely include schemata to identify the interfaces between the various layers of the system, facilitate transporting and maintaining the system, and formally defining the user interface with the database.

Like most of my projects at VPI&SU, the database system was a one-person operation. In the "real world", however, a database system would be a large revenue-generating project that would be done by a group of people in a specified timeframe. As such, the success of the project (and for a small operation, the success of the entire company), would hinge on

the adherence to a well-defined design methodology. Unfortunately, I did not realize this until after I had entered the testing phase of this project, where I felt obligated to resist the urge to scrap the previous 2 years of coding and start again from scratch.

Ten page buffers was not enough to execute some of the more complex database commands. Too frequently, a desired page that was thought to be in a page buffer, had been paged out due to a subsequent operation such as searching a large B-tree with lots of pages. The end result was that the database program had to be changed (in all 9 copies) to reaccess the desired page.

There are four potential approaches to solving this problem:

- A mechanism that temporarily locked specific pages in a buffer would resolve this problem, but would thwart the LRU paging algorithm by reducing the number of page buffers available. Some approach like DB2 (locking and releasing during updates only) might relieve this problem.
- Every access to data in a page counts towards the LRU algorithm. This might correct the locking problem, but would quickly turn into a performance problem.
- Have two sets of buffers: locked and pageable. The locked set would include the RELCATALOG information for select system relations, and any temporary pages that are needed. The pageable set would follow the LRU algorithm. This approach of having transient and non-transient buffers would require searching both the locked and pageable queues every time an access is made to the page buffers.
- Have more than ten page buffers. The need to reaccess specific page buffers did not manifest itself when twenty buffers were used. Commercial database systems such as IBM's DB2 typically have many pools of buffers, with each pool satisfying different requirements, such as a pool of large buffers to handle bulk data transfer.

To improve the efficiency of B-trees, larger pages should be used. Although the 256-byte combinations did not show significant degradation because of the smaller page size, this could easily change as a system gets busier. This observation becomes obvious when the space utilization of B-tree nodes is considered:

For each page,

- 24 bytes are used for page pointer overhead (see Figure 3.1 on page 7 for details),

For each B-tree node tuple on a page,

- 4 bytes are occupied by the tuple's offset slot,
- 4 bytes are used to hold the number of keys in the node,

For each B-tree key on the page,

- 4 bytes are used to point to the the link node tuple heading the list of TIDs of tuples with the specified key,
- 4 bytes are used for the subtree pointer,
- 4 bytes are used for the key length,
- 24 bytes are used for block overhead (see Figure 3.2 on page 75) since each key is allocated from the workspace area,
- an arbitrary number of bytes are required to hold the actual B-tree key.

For 256-byte pages, very little space is actually being used to hold the desired information: the B-tree key and DATATID pointer.

In general, the database system was too slow. A real database system would be in assembler language or a high-level language with a good optimizer that allows use of assembler language routines to perform efficient I/O. Although this was my first PASCAL program, it was easy to observe that PASCAL is not an ideal language for implementing large systems. The data type restrictions tended to force hokie approaches to data management that destroyed system performance. For example, the problem with having the same data area represent both character and

integer data resulted in conversion subroutines that had to be run every time an integer was put into or retrieved from a page buffer. An elimination of these conversions alone should greatly improve performance.

In many computer installations, the operation and maintenance of a large database is the main function of the computer center. In such an environment, the pages containing the database system programs and the page buffers would be locked (fixed) in memory, eliminating these pages from consideration in the operating system's paging algorithms. The execution priority of the database system would be much higher and operating system routines would likely be used to perform such mundane functions as allocating blocks of storage. Such improvements would definitely result in improved system performance.

Ordered relations performed better than unordered relations in terms of execution time and space utilization for just about all database commands tested. This is because the need to access and update one or more separate B-trees was eliminated in ordered relations by having the NEXTTID field within the tuple itself. Once relational operators like JOIN and SELECT have been implemented, additional tests need to be run to determine whether ordered relations retain their performance advantage. A prediction at this point would be that ordered relations are excellent for sequential operations like printing a relation, but performance would suffer greatly if ordered relations were used to perform relational operations or process complex database queries.

Many more tests need to be run

- to identify what sections of database system code were executed most often. These sections should be optimized.

- with more page buffers (perhaps 40, 60, 80, 100, and 200). Thirty page buffers is far too few buffers for a real database system.
- with longer runs than 50 command executions. The number of runs should be large enough that statistical analysis could be performed to identify whether the measurements obtained are accurate and representative.

The statistics presented in Chapter 10 yielded very few conclusions concerning database performance. One phenomenon the statistics failed to explain was why were so many system page faults observed with the 1024-30 combination in Tests 1-4, and not in similar configurations such as 1024-10 and 512-30? Most likely this was because with thirty 1024-byte pages locked in the working set, 40% of the default VMS working set (150 512-byte pages) was being used to hold page buffers. This meant that the remaining 60% of the working set had to hold both the remaining data, plus the database system program. System page faults were constantly being taken because the database program was thrashing, not the data!

Suggestions for Future Work

This section suggests directions for future modifications to enhance the database. The proposals discussed in "Database Security" on page 191 and "Database Recovery" on page 192 should also be considered.

Changes to Database Physical Structure

- Too many database system modules had to be concerned with storage management (i.e. guaranteeing that storage obtained from the

Workspace Area was released when it was no longer needed). A solution would be to maintain a list of storage allocated by a specific command. When command execution finished, storage would be freed -- except for certain long term storage requirements such as ARL entries.

- There was no need for variable-length tuples in this project. Potentially this requirement could arise in the future. If so, a mechanism needs to be found to relocate a tuple to another page if the larger tuple did not fit on its current page. The existing offset slot approach already suffices if space for a tuple is found on the same page. System R uses a similar offset slot approach and also supports variable-length tuples.
- Having 9 separate copies of the database system soon proved to be a problem every time an error was encountered in testing. Ideally, the page size for a specific database should be defined when the database is created, and obtained from page 1 of the database when the database is first accessed. Unfortunately, VMS does not allow one to read and write from a page if the page size is not known. Perhaps some global database directory could be kept across the entire VMS system that identifies the page size applicable to each spatial database system.
- When a tuple in a relation refers to another relation or spatial data structure, the tuple contains the name of the other structure. An alternative that would improve the efficiency immensely would be to replace the name with the Relational Catalog TID of the structure being referred to. This approach would eliminate the intermediate Relational Catalog references.
- If the database program is ever to be used by more than one user at one time, a lock hierarchy is needed to provide for concurrent access to ARL entries, page buffers, and the various system queues. Pages being modified would have to be locked until all information on the page has been committed, or aborted.

- Page 1 of a spatial database contains system variables and is always written to disk, even if no database changes occurred. The system should be modified to write page 1 only if the system variables at DB_DOWN time do not match their original values.
- Maintain a use count for relations, SDSs, and prototypes. This effort was avoided for this project since it would greatly slow down updating tuple fields (e.g. if the fields were names of relations). An alternative might be to maintain the use count information for all relations on Page 1 of the database (or in some new system relation, accessable by relation name), writing the values to disk only when the database was brought down.

Changes to Database Logical Structure

- There is no firm boundary between the various layers of the database program. The system modules should be restructured so that there is a boundary with a definable interface between modules that interact with the user, modules that operate on a relation level (i.e. operate on ARL entries), and modules that are concerned with pages and buffers.
- Expand Relational Prototypes to identify default values for all fields. This would greatly simplify creating a relation where a large percentage of the tuples always have the same value in a specific field (e.g. MINIMUM_DRINKING_AGE = 21).
- Expand Relational Prototypes to identify permissible fields as an alternative to specifying minimum and maximum values. If a field can only take on the values 1, 6, and 23, specifying a range of 1-23 will not insure the integrity of the field.
- Unordered relations were much slower than their ordered counterparts, in part because an independent B-tree had to be maintained for every change to the unordered relation. The B-tree was needed because some field had to be used for tuple ordering. An alternative would be to add a NEXTTID field to unordered relations.

This field would be used unless some alternate ordering was specified. With such an approach, an accessing B-tree would only be created for indexed fields; unordered relations with no indexed fields would perform the same as ordered relations. The end result would be that ordered relations would be a subset of unordered relations.

- Specifying that an indexed field in an unordered relation is unique is one mechanism of ensuring that all values in a specific tuple field are unique. Unfortunately, this approach does not work for ordered relations since there are no indexed fields (or accessing B-trees). Some alternate mechanism is needed to provide a means of ensuring field uniqueness in an ordered relation. The suggestion in the previous paragraph might be one solution.
- Each SDS defined in a geographical information system has an AV relation which never seems to have more than one tuple. For example, a STATE SDS will have a STATE_AV relation which defines, in one "fat" tuple, the state's area, population, governor, major crop, etc. Such information is wasteful with the current implementation since one page is allocated for the AV relation, even though it will never have more than one tuple. Three solutions are available. First, change the prototype for the AV relation to define each field as a single tuple. For example, area would be one tuple, while major crop would be another. Another solution would be to somehow consolidate all of this information for all SDSs in one relation and have each SDS have a pointer to the AV tuple that applies. A third approach would be to somehow include this information in either the Relational Catalog tuple for the SDS, or within the SDS relation itself.
- The atom POINT should be implemented, perhaps as a pointer to a X,Y pair. Currently a X,Y pair is represented by two fields in a tuple, but with an indivisible pair, the two values would always be linked together.

Changes to Interpreter File Structure

In the RDSVOCB.DAT file that defines the Interpreter Vocabulary, also define the required number of stack elements and the characteristics of those elements (e.g. TOS-2 must be character). This would permit the interpreter to centralize error validation and would guarantee that the same error messages are issued for similar problems. Adding additional tests applicable to all interpreter commands would also be simplified since the error checking would all be done in one location.

Changes to Interpreter Commands

The following types of commands should be added to the query language interpreter:

- A command to write a message to the current output file without having to add the message to the stack, issue the period (.) command to write the message, and then issue the DROP command to delete the message from the interpreter stack.
- A command to generate a unique name which can be used as the name of a temporary relation or variable.
- Commands to add and delete field definitions in a prototype already in use. This would require keeping track of what relations use a specific prototype and rebuilding those relations according to the new definition.
- A command to rename a relation. This would involve accessing the Relational Catalog tuple for the relation to be renamed, and changing the RELNAME field to the new relation name. Since the Relational Catalog relation is unordered, the accessing B-tree will automatically be updated to reflect the change in the relation name.
- Union, Intersection, Join, Projection, and other relational operators should be implemented first as user vocabulary commands, and then perhaps as direct calls to the appropriate system subroutines (to avoid the stack integrity checking). All of the facilities already exist to develop these commands. For example,

UNION would 1) verify that both relations have the same prototype, 2) create a new relation with the same prototype to represent the output, 3) access both relations, 4) step through the two relations using the NEXT_TUPLE and ENDLIST? interpreter commands, and 5) use the relational operators (>, =, etc.) and the COPY_TUPLE command to merge the two relations and eliminate duplicates.

- Implement a BOOL_ELEM stack element. This could be used to provide some integrity when a command can only have a boolean input (e.g. UNTIL and IF statements).
- Commands to retrieve information from the Relational Catalog tuple for a relation. Currently only the name of the relation's prototype can be retrieved. Potentially, the number of tuples in the relation and the relation type might also be valuable.

Changes to User Interface

- A user of the interpreter should not have to know that the system is based on a stack. Instead, a high-level language should be developed that permits specification of simple operations like 2 + 3? and complex operations like "Which rivers are within 50 miles of Raleigh, N.C.?". The interpreter language should be easy to user, and oriented to non computer-oriented users (another reason to mask the interpreter stack).
- Next, user access to a spatial database should be via a full-screen mechanism. This would simplify bulk data entry. Also, more than one tuple in a relation could be displayed and/or modified at once.
- Finally, a database user should not have to be concerned with the underlying physical structure of the database, only the names of the relations, prototypes, and spatial data structures he or she wants to access. The first access of a relation should create an Active_Relationlist entry. Subsequent entries should use the entries to avoid excessive disk references. A Least-Recently-Used

algorithm should be put into place to eventually delete an ARL entry for a relation.

BIBLIOGRAPHY

[1] Astrahan, M.M., M.W. Blasgen, D.D Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L Traiger, B.W. Wade, V. Watson, "System R: A Relational Approach to DataBase Management". ACM Transactions on Database Systems Vol. 1, No. 2 (June 1976), pp. 97-137.

[2] Blasgen, M.W., Gray, J., Mitoma, M., Price, T., "The Convoy Phenomenon". Operating Systems Review Vol. 13, No. 2 (April 1979), pp. 20-25.

[3] Chamberlin, D.D., M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B.G. Lindsay, R.A. Lorie, J.W. Mehl, T.G. Price, G.R. Putzolu, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, R.A. Yost, "A History and Evaluation of System R". Communications of the ACM Vol. 24, No. 10 (Oct 1981), pp. 632-646.

[4] Chamberlin, D.D., R.F. Boyce, "SEQUEL: A Structured English Query Language". Proc. ACM-SIGFIDET Workshop (May 1974), pp. 249-264.

[5] Chamberlin, D.D., J.N. Gray, I.L. Traiger, "Views, Authorization, and Locking in a Relational Data Base System", AFIPS Conf. Proc. 1977 NCC, Vol 44, (1977), pp. 425-430

[6] Cheng, J.M., C.R. Loosley, A. Shibamiya, P.S. Worthington, "IBM Database 2 Performance: Design, Implementation, and Tuning", IBM Systems Journal Vol 23, No 2 (1984), pp. 189-210.

[7] Claybrook, B.G., File Management Techniques, John Wiley & Sons, 1983.

[8] Cohill, A., R. Lundin, "The Great New Mini Database", Department of Computer Science, VPI&SU, Blacksburg, VA 24061, April 1982.

[9] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM 13,6 (June 1970), pp. 377-387.

[10] Crus, R.A., "Data Recovery in IBM Database 2", IBM Systems Journal Vol 23, No 2 (1984), pp. 178-188.

[11] Dash, J.R., R.N. Ojala, "IBM Database 2 in Information Management System Environment", IBM Systems Journal Vol 23, No 2 (1984), pp. 165-177.

[12] Date, C.J., An Introduction to Data Base Systems. Addison-Wesley Publishing, USA, 1977.

[13] Date, C.J., An Introduction to Data Base Systems, Volume 2 Addison-Wesley Publishing, USA, 1983.

[14] Date, C.J., A Guide to DB2, Addison-Wesley Publishing, USA, 1984.

[15] Dodd, G.G., "Elements of Data Management Systems", Computing Surveys Vol 1, No 2 (June 1969), pp. 117-133.

[16] Goldstein, R.C., A.J. Strnad, "The MacAIMS Data Management System", Proceedings of the 1970 ACM-SIGFIDET Workshop on Data Description and Access (April 1971).

[18] Griffiths, P.P., B.W. Wade, "An Authorization Mechanism for a Relational DataBase System". ACM Transactions on Database Systems Vol. 1, No. 3 (Sept 1976), pp. 242-255.

[17] Fernandez, E.B., R.C. Summers, C. Wood, Database Security and Integrity, Addison-Wesley Publishing, USA, 1981.

[19] Haderle, D.J., R.D. Jackson, "IBM Database 2 Overview", IBM Systems Journal Vol 23, No 2 (1984), pp. 112-125.

[20] Hawthorn, P.B., D.J. DeWitt, "Performance Analysis of Alternative Database Machine Architectures", IEEE Transactions on Software Engineering, Vol SE-8, No 1, (January 1982), pp. 61-75.

[21] Held, C.D., M.R. Stonebraker, E. Wong, "INGRES: A Relational Database System", Proc. ACM Pacific 75 Regional Conf., Vol 44, (May 1975), pp. 409-416.

[22] Held, G.D., M.R. Stonebraker, E. Wong, "INGRES - A Relational Data Base System". Proc. 1975 AFIPS Conference, March 1975, pp. 409-416.

[23] Horowitz E. and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Inc., Potomac MD 20852, 1977.

[24] Hsiao, D.K., Advanced Database Machine Architecture, Prentice-Hall, NJ (1983).

[25] Hutt, A.T.F., A Relational Data Base Management System, John Wiley & Sons, New York, 1979.

[26] Kahn, S., "An Overview of three Relational Database Products". IBM Systems Journal Vol. 23, No. 2 (1984), pp. 100-111.

[27] Krusemark, S., R.M. Haralick, "Operating System Interface", Technical Report # SDA 81-1, SDA Lab, VPI&SU, Blacksburg VA 24061, April 1981.

[28] Liles, W.C., E.D. Nugent, "Data Management For Large Cartographic Information Systems", Technology Service Corporation, Santa Monica, CA 90405

[29] McLeod, D.J., M.J. Meldman, "RISS - A Generalized Minicomputer Database Management System", AFIPS Conf. Proc. 1977 NCC, Vol 44, (1977), pp. 397-402

[30] Minden, G., A systems Manual for a Query Language Interpreter, Unpublished Project Report, University of Kansas, Lawrence, Kansas.

[31] Mylopoulous, J., S. Schuster, D. Tsichritzis, "A Multi-Level Relational System", AFIPS Conf. Proc. 1977 NCC, Vol 44, (1977), pp. 403-408

[32] Notley, M.G., "The Peterlee IS/1 System", UKSC 18, March 1972.

[33] Parker, J., "File Structures and Access Methods for a Geographic Information Retrieval System", Proceedings of the 1972 ACM-SIGFIDET Workshop on Data Description, Access, and Control, (Dec 1972).

[34] Rothnie, J.B. Jr., "An Approach to Implementing a Relational Data Management System", ACM-SIGMOD Workshop on Data Description, Access, and Control, (May 1974), pp. 277-294.

[35] Rothnie, J.B. Jr, "Evaluating Inter-Entry Retrieval Expressions in a Relational Database Management System", AFIPS Conf. Proc. 1977 NCC, Vol 44, (1977), pp. 417-423.

[36] Schmidt, J.W., M.L. Brodie, Relational Database Systems, Springer-Verlag (1983).

[37] Shapiro, L.G., R.M. Haralick, "A Spatial Data Structure", Technical Report # CS 79005-R, Department of Computer Science, VPI&SU, Blacksburg, VA 24061, August 1979.

[38] Shapiro, L.G., R.M. Haralick, "A Spatial Data Structure", Geo-Processing, 1 (1980), pp. 313-337.

[39] Shapiro, L.G., S. Engineer, "A Query Language for a Spatial Information System", Proceedings of the XV Congress on Photogrammetry and Remote Sensing, June 1984.

[40] Stonebraker, M.R., "Implementation of Integrity Constraints and Views by Query Modification". Proc. 1975 ACM-SIGMOD International Conference on the Management of Data (MAY 1975).

[41] Stonebraker, M.R., E. Wong, P. Kreps, G.D. Held, "The Design and Implementation of INGRES", ACM Transactions on DataBase Systems 1, No. 3 (September 1976), pp. 189-221.

[42] Stonebraker, M.R., E. Wong, "Access Control in a Relational Database Management System by Query Modification", Proc ACM National Conference, 1974, pp. 180-187.

[43] Stonebraker, M.R., P. Rubinstein, "The INGRES Protection System", Proc ACM National Conference, 1976, pp. 80-84.

[44] Strnad, A.J., "The Relational Approach to the Management of Databases", Proceedings of IFIP Congress 71, Vol 2 (1971), pp. 901-904.

[45] Todd, S.J.P., "The Peterlee Relational Test Vehicle - A System Overview", IBM Systems Journal 15:4 (1976), pp. 285-308.

[46] Todd, S., "PRTV: An Efficient Implementation For Large Relational Data Bases", Proceedings International Conference on Very Large Data Bases, (Sept 1975).

[47] Ullman, J.D., Principles of DataBase Systems. Computer Science Press, USA, 1980.

[48] Vaidya, P.D., L.G. Shapiro, R.M. Haralick, G.J. Minden, "Design and Architectural Implications of A Spatial Information System", Technical Report # CS 820003-R, Department of Computer Science, VPI&SU, Blacksburg VA 24061, December 1981.

[49] Vaidya, P., "An Experimental Spatial Information System", VPI&SU Computer Science Department, Blacksburg VA 24061, Jan 1982.

[50] Whitney, V.K.M., "Relational Data Management Implementation Techniques", Proceedings of the ACM-SIGMOD Workshop on Data Description, Access, and Control, (May 1974), pp. 321-350.

[51] Wiederhold, G., Database Design, McGraw-Hill, NM, 1977.

[52] Zloof, M.M., "Query-By-Example: A Database Language", IBM System Journal 16:4, 1977.

[53] Zloof, M.M., "Query-By-Example: A Database Language", AFIPS Conf. Proc. NCC, Vol 44, (1975), pp. 431-438.

[54] An Introduction to SQL, GC26-4082, IBM Corporation, Armonk, NY.

[55] Query-By-Example Program Description/Operating Manual SH20-2077, IBM Corporation, Armonk, NY.

APPENDIX A - SAMPLE INTERPRETER COMMAND FILE

As described in "Interpreter Command File" on page 79, each interpreter command has 3 fields: 1) a command precedence field, 2) a CASE statement label number, and 3) the command name. The function of each command is described in "Chapter 5 - The Query Language Interpreter" on page 101.

Listed below are the contents of RDSVOCB.DAT, the interpreter command file:

```
2 1 CONSTANT
2 2 VARIABLE
0 3 FETCH
0 4 STORE
0 5 +
0 6 -
0 7 *
0 8 /
0 9 .
0 11 (STACK)
0 12 INPUT>
0 13 ***EOF
2 14 DEFINITION
3 15 END_DEF
0 16 ALLOCATE
0 17 IDUMP
0 20 NOT
2 21 DO
2 22 +LOOP
2 23 IF
2 24 THEN
2 25 ELSE
2 26 REPEAT
2 27 UNTIL
0 30 #STACK
0 36 I
0 37 J
0 38 K
0 39 L
0 41 =
0 42 ¬=
0 43 >=
0 44 <=
0 45 >
0 46 <
0 47 SWAP
0 48 DROP
0 49 DUP
0 50 -ROT
0 51 ROT
0 52 OVER
0 53 DONE
0 55 >OUTPUT
0 58 "
0 59 .STACK
```

```
0 60 INT?
0 61 UDICT?
0 62 CHAR?
0 63 IDICT?
0 66 #DROP
0 67 #ROT
0 68 #-ROT
0 69 FORGET
0 70 #UVOCB
0 71 ARL?
0 74 PRINTFREE
0 75 DB_CREATE
0 76 DB_UP
0 77 HELP
0 78 #IVOCB
0 79 UDUMP
0 80 DONEOK
0 81 &
0 82 |
0 83 BREAK
0 84 NEXT
0 85 PRINTON
0 86 PRINTOFF
0 87 SSTATS
0 88 CLEAR
0 100 DELETE_SDS
0 101 LISTREL
0 102 CREATE_RELPROTO
0 103 ADDTO_RELPROTO
0 104 ENDLIST?
0 105 #TUPLES
0 106 ACCESSREL
0 107 RELEASEREL
0 108 CREATEREL
0 109 DELETEREL
0 110 DELETE_PROTO
0 111 CREATE_SDSPROTO
0 112 ADDTO_SDSPROTO
0 113 BUILD_TUPLE
0 114 PRINT_TUPLE
0 115 ATTACHREL
0 116 PRINTREL
0 117 PRINTSDS
0 118 REMOVEREL
0 119 ACCESS_REL
0 120 PRINTREL_SDS
0 121 NEXT_TUPLE
0 122 ACCESSREL_FN
0 123 PRINTREL_FN
0 124 PRINT#ARL
0 125 SETVALUE
0 126 DELETE_TUPLE
0 127 PRINTBRT
0 128 PROTOTYPE
0 131 DB_DOWN
0 132 #RELS
0 133 #SDS
0 134 #PROTOS
0 135 PRINTARL
0 137 STATS
0 138 VALUE_OF
0 139 CREATE_SDS
0 140 DELETE_SDS
0 141 COPY_TUPLE
```

APPENDIX B - SAMPLE INTERPRETER VOCABULARY FILE

The interpreter commands listed below could occur in an interpreter vocabulary file. These commands

- define a vocabulary constant named MAXINT that has a value of 2147483647, the maximum integer value permitted by PASCAL,
- define a vocabulary constant named MININT that has a value of -2147483647, the minimum integer value permitted by PASCAL, and

```
2147483647
" MAXINT
CONSTANT

-2147483647
" MININT
CONSTANT
```

APPENDIX C - SAMPLE USER VOCABULARY FILE

The interpreter commands listed below could occur in a user vocabulary file. These commands define a new vocabulary command, #OR, that performs an OR operation (| command) on a specified number of stack elements. #OR could be used, for example, just before an UNTIL statement to OR together a group of loop conditions.

```
#OR
DEFINITION
  #STACK
  2
  <
  IF
    " At least 2 elements required for #OR
    .
    DROP
  ELSE
    INT?
    0
    =
    IF
      " Count for #OR must be integer
      .
      DROP
    ELSE
      DUP
      1
      <=
      IF
        " Count for #OR must be > 1
        .
        DROP
      ELSE
        " NBR
        CONSTANT
        #STACK
        NBR
        <
        IF
          " Not enough elements for #OR
          .
          DROP
        ELSE
          1
          NBR
          1
          -
          DO
            |
            DONEOK
            0
            =
            IF
              " Terminating #OR due to error
              .
              DROP
              BREAK
```

```
            THEN
          +LOOP
          " NBR
          FORGET
        THEN
      THEN
    THEN
  THEN

END_DEF
```

APPENDIX D - SAMPLE HELP FILE

Listed below is the HELP file for the + command, HELP005.HLP:

```
(*****************************************************************)
(* Command:         +                                            *)
(*                                                               *)
(* Function:        Add the top two stack elements.  The result  *)
(*                  replaces the input operands on the stack.    *)
(*                                                               *)
(* Stack Before:    Position        Description        Data Type *)
(*                  --------     -----------------     --------- *)
(*                     TOS           <integer1>          INT_ELEM *)
(*                   TOS-1           <integer2>          INT_ELEM *)
(*                                                               *)
(* Stack After:     Position        Description        Data Type *)
(*                  --------     -----------------     --------- *)
(*                     TOS      <integer1>+<integer2>    INT_ELEM *)
(*                                                               *)
(* Example:                                                      *)
(*                                                               *)
(*       Interpreter Command:        +                           *)
(*                                                               *)
(*       Stack Before:               4                           *)
(*                                   3                           *)
(*                                                               *)
(*       Stack After:                7                           *)
(*                                                               *)

(*****************************************************************)
```

APPENDIX E - SAMPLE VOCABULARY COMMANDS

This appendix defines sample vocabulary commands. The execution of each command, implemented as vocabulary words, would be fairly slow since the interpreter must perform integrity checks of the input stack elements for every interpreter command executed. Also, the function of many of the commands in the vocabulary words is only to relocate elements on the interpreter stack, instead of performing meaningful functions. In a real database environment, the commands would be implemented in the same programming language as the rest of the database programs (in this case, PASCAL), performing integrity checks once at the start of the command processing. The following vocabulary commands are defined:

- The COPYREL command produces an identical copy of an existing relation.
- The UNION command creates a new relation that is the union of two relations.
- The INTERSECT command creates a new relation that is the intersection of two relations.
- The FINDFLDS command creates a new relation containing copies of all tuples in the source relation that have specific field values.

In all of the vocabulary words to follow, the comments are only to clarify the sample vocabulary commands; they would not be permitted if the vocabulary words were implemented as shown.

The following shorthand notation is used to illustrate the contents of the interpreter stack after a command is executed:

```
command      <TOS> <TOS-1> <TOS-2>
```

For example, if the interpreter stack contains the elements TOS=T1, TOS-1=T2, TOS-2=T3, the sequence

```
SWAP        <T2> <T1> <T3>
```

means that after executing the SWAP command, T2 is at the top of the interpreter stack, followed by T1 and T3.

The following sequence, entered top-down, uses the new vocabulary words to create a new relation T that consists of all tuples satisfying the criteria:

T = (A UNION B) INTERSECT (Tuples in C with Field F1=K and Field F2=L)

Note that two temporary relations must be used.

```
" B
" A
" TMP1
UNION
" L
" F2
" K
" F1
2
" C
" TMP2
FINDFLDS
" TMP1
" TMP2
" T
INTERSECT
" TMP1
DELETEREL
" TMP2
DELETEREL
```

Command: COPYREL

Function: Produce a new relation that is a copy of an existing relation.

Stack Before:

Position	Description	Data Type
TOS	Name of target (TGT)	CHAR_ELEM
TOS-1	Name of source (SRC)	CHAR_ELEM
TOS-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

```
" COPYREL
DEFINITION

  #STACK                          <# elements><TGT><SRC>
  2                               <2><# elements><TGT><SRC>
  >=                              <T/F><TGT><SRC>
  IF                              At least 2 stack elements?
    DUP                           <TGT><TGT><SRC>
    ?CHAR                         <T/F><TGT><SRC>
    IF                            Is TGT name character?
      SWAP                        <SRC><TGT>
      DUP                         <SRC><SRC><TGT>
      ?CHAR                       <T/F><SRC><TGT>
      IF                          Is SRC name character?
        DUP                       <SRC><SRC><TGT>
        PROTOTYPE                 <SRC Prototype><SRC><TGT>
        ROT                       <TGT><SRC Prototype><SRC>
        DUP                       <TGT><TGT><SRC Prototype><SRC>
        -ROT                      <TGT><SRC Prototype><TGT><SRC>
        CREATEREL                 Create TGT relation
        DONEOK                    <T/F><TGT><SRC>
        IF                        Could TGT be created?
          ACCESSREL               <TGT ARL><SRC>
          DONEOK                  <T/F><TGT ARL><SRC>
          IF                      Could TGT be accessed?
            SWAP                  <SRC><TGT ARL>
            ACCESSREL             <SRC ARL><TGT ARL>
            DONEOK                <T/F><SRC ARL><TGT ARL>
            IF                    Could SRC be accessed?
              ENDLIST?            <T/F><SRC ARL><TGT ARL>
              IF                  EOF <SRC>?
                " SOURCE RELATION IS EMPTY
                .
                DROP
              ELSE                Not EOF <SRC>?
                REPEAT            Loop until EOF<SRC>
                  SWAP            <TGT ARL><SRC ARL>
                  COPY_TUPLE      Copy SRC tuple to TGT relation
                  SWAP            <SRC ARL><TGT ARL>
                  NEXT_TUPLE      Next tuple in SRC
```

```
                ENDLIST?          EOF <SRC>?
              UNTIL               Loop until EOF<SRC>
            THEN
            RELEASEREL            Release ARL for SRC relation
          ELSE
            " COULD NOT ACCESS SOURCE RELATION
            .
            DROP
          THEN
          RELEASEREL              Release ARL for TGT relation
        ELSE
          " COULD NOT ACCESS RELATION JUST CREATED
          .
          DROP
        THEN
      ELSE
        " COULD NOT CREATE NEW RELATION
        .
        DROP
      THEN
    ELSE
      " ??? MUST BE CHARACTER
      .
      DROP
    THEN
  ELSE
    " ?? MUST BE CHARACTER
    .
    DROP
  THEN
ELSE
  " COPYREL REQUIRES AT LEAST 2 STACK ELEMENTS
  .
  DROP
THEN

END_DEF
```

Command: UNION

Function: Create a new relation that is the union of two existing relations.

Stack Before:

Position	Description	Data Type
TOS	Name of target (TGT)	CHAR_ELEM
TOS-1	Name of source #1 (S1)	CHAR_ELEM
TOS-2	Name of source #2 (S2)	CHAR_ELEM
TOS-3	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

```
" UNION
DEFINITION

  #STACK                                  <# elements><TGT><S1><S2>
  3                                       <3><# elements><TGT><S1><S2>
  >=                                      <T/F><TGT><S1><S2>
  IF                                      At least 3 elements on stack?

    " @@TARGET                            <"@@TARGET"><TGT><S1><S2>
    CONSTANT                              Constant for name of Target
    " @@SOURCE1                           <"@@SOURCE1"><S1><S2>
    CONSTANT                              Constant for name of Source 1
    " @@SOURCE2                           <"@@SOURCE2"><S2>
    CONSTANT                              Constant for name of Source 2
    @@TARGET                              <TGT>
    PROTOTYPE                             <TGT proto>
    DONEOK                                <T/F><TGT proto>
    NOT                                   <F/T><TGT proto>
    IF                                    Does target relation exist?
      @@SOURCE1                           <S1>
      PROTOTYPE                           <S1 proto>
      DONEOK                              <T/F><S1 proto>
      IF                                  Does S1 relation exist?
        DUP                               <S1 proto><S1 proto>
        @@SOURCE2                         <S2><S1 proto><S1 proto>
        PROTOTYPE                         <S2 proto><S1 proto><S1 proto>
        DONEOK                            <T/F><S2 pro><S1 pro><S1 pro>
        IF                                Does S1 relation exist?
          =                               <T/F><S1 proto>
          IF                              Do S1 and S2 have same proto?
            @@TARGET                      <TGT><S1 proto>
            CREATEREL                     Create TGT relation
            DONEOK                        <T/F>
            IF                            TGT created successfully?
              @@TARGET                    <TGT>
              ACCESSREL                   <TGT ARL>
              DONEOK                      <T/F><TGT ARL>
              IF                          Could access TGT relation?
                @@SOURCE1                 <S1><TGT ARL>
```

```
ACCESSREL                  <S1 ARL><TGT ARL>
DONEOK                     <T/F><S1 ARL><TGT ARL>
IF                         Could access S1 relation?
  @@SOURCE2                <S2><S1 ARL><TGT ARL>
  ACCESSREL                <S2 ARL><S1 ARL><TGT ARL>
  DONEOK                   <T/F><S2 ARL><S1 ARL><TGT ARL>
  IF                       Could access S2 relation?
    REPEAT                 Loop until EOF <S1 and S2>
      ENDLIST?             <T/F><S2 ARL><S1 ARL><TGT ARL>
      IF                   EOF <S2> ?
        SWAP               <S1 ARL><S2 ARL><TGT ARL>
        ENDLIST?           <T/F><S1 ARL><S2 ARL><TGT ARL>
        NOT                <F/T><S1 ARL><S2 ARL><TGT ARL>
        IF                 Not EOF <S1>
          ROT              <TGT ARL><S1 ARL><S2 ARL>
          SWAP             <S1 ARL><TGT ARL><S2 ARL>
          REPEAT           Loop through rest of S1
            SWAP           <TGT ARL><S1 ARL><S2 ARL>
            COPY_TUPLE     Copy S1 tuple to TGT relation
            SWAP           <S1 ARL><TGT ARL><S2 ARL>
            NEXT_TUPLE     Obtain next tuple in S1
            ENDLIST?       EOF <S1> ?
          UNTIL            Loop through rest of S1
          ROT              <S2 ARL><S1 ARL><TGT ARL>
        THEN
      ELSE                 Not EOF<S2>
        SWAP               <S1 ARL><S2 ARL><TGT ARL>
        ENDLIST?           <T/F><S1 ARL><S2 ARL><TGT ARL>
        IF                 EOF <S1> ?
          SWAP             <S2 ARL><S1 ARL><TGT ARL>
          ROT              <TGT ARL><S2 ARL><S1 ARL>
          SWAP             <S2 ARL><TGT ARL><S1 ARL>
          REPEAT           Loop through rest of S2
            SWAP           <TGT ARL><S2 ARL><S1 ARL>
            COPY_TUPLE     Copy S2 tuple to TGT relation
            SWAP           <S2 ARL><TGT ARL><S1 ARL>
            NEXT_TUPLE     Obtain next tuple in S2
            ENDLIST?       EOF <S2> ?
          UNTIL            Loop through rest of S2
          SWAP             <TGT ARL><S2 ARL><S1 ARL>
          -ROT             <S2 ARL><S1 ARL><TGT ARL>
        ELSE               Not EOF<S1> or EOF<S2>
          SWAP             <S2 ARL><S1 ARL><TGT ARL>
          >                <T/F><S2 ARL><S1 ARL><TGT ARL>
          IF               Is S1 tuple > S2 tuple?
            ROT            <TGT ARL><S2 ARL><S1 ARL>
            COPY_TUPLE     Copy S2 tuple to TGT relation
            -ROT           <S2 ARL><S1 ARL><TGT ARL>
          ELSE             S1 tuple <= S2 tuple
            SWAP           <S1 ARL><S2 ARL><TGT ARL>
            ROT            <TGT ARL><S1 ARL><S2 ARL>
            COPY_TUPLE     Copy S1 tuple to TGT relation
            -ROT           <S1 ARL><S2 ARL><TGT ARL>
            SWAP           <S2 ARL><S1 ARL><TGT ARL>
          THEN
        THEN
      THEN
      SWAP                 <S1 ARL><S2 ARL><TGT ARL>
      ENDLIST?             EOF <S1> ?
      ROT                  <S2 ARL><T/F><S1 ARL><TGT ARL>
      ENDLIST?             EOF <S2> ?
      ROT                  <T/F><T/F><S2><S1><TGT ARL>
      AND                  <T/F><S2 ARL><S1 ARL><TGT ARL>
    UNTIL                  Loop until EOF <S1 and S2>
    RELEASEREL             Release ARL for S2 relation
  ELSE
```

```
                    " COULD NOT ACCESS SOURCE RELATION #2
                    .
                    DROP
                  THEN
                  RELEASEREL                Release ARL for S1 relation
                ELSE
                  " COULD NOT ACCESS SOURCE RELATION #1
                  .
                  DROP
                THEN
                RELEASEREL                  Release ARL for TGT relation
              ELSE
                " COULD NOT ACCESS TARGET RELATION
                .
                DROP
              THEN
            ELSE
              " COULD NOT CREATE TARGET RELATION
              .
              DROP
            THEN
          ELSE
            " SOURCE RELATIONS MUST HAVE SAME PROTOTYPE
            .
            DROP
          THEN
        ELSE
          " SOURCE #2 DOES NOT EXIST
          .
          DROP
        THEN
      ELSE
        " SOURCE #1 DOES NOT EXIST
        .
        DROP
      THEN
    ELSE
      " THE TARGET RELATION ALREADY EXISTS
      .
      DROP
    THEN

    @@SOURCE1                               Delete constant for S1
    FORGET
    @@SOURCE2                               Delete constant for S2
    FORGET
    @@TARGET                                Delete constant for TGT
    FORGET

  ELSE
    " UNION COMMAND REQUIRES AT LEAST 3 STACK ELEMENTS
    .
    DROP

END_DEF
```

Command: INTERSECT

Function: Create a new relation that is the intersection of two existing relations.

Stack Before:

Position	Description	Data Type
TOS	Name of target (TGT)	CHAR_ELEM
TOS-1	Name of source #1 (S1)	CHAR_ELEM
TOS-2	Name of source #2 (S2)	CHAR_ELEM
TOS-3	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

```
" INTERSECT
DEFINITION

  #STACK                                <# elements><TGT><S1><S2>
  3                                     <3><# elements><TGT><S1><S2>
  >=                                    <T/F><TGT><S1><S2>
  IF                                    At least 3 elements on stack?

    " @@TARGET                          <"@@TARGET"><TGT><S1><S2>
    CONSTANT                            Constant for name of Target
    " @@SOURCE1                         <"@@SOURCE1"><S1><S2>
    CONSTANT                            Constant for name of Source 1
    " @@SOURCE2                         <"@@SOURCE2"><S2>
    CONSTANT                            Constant for name of Source 2
    @@TARGET                            <TGT>
    PROTOTYPE                           <TGT proto>
    DONEOK                              <T/F><TGT proto>
    NOT                                 <F/T><TGT proto>
    IF                                  Does target relation exist?
      @@SOURCE1                         <S1>
      PROTOTYPE                         <S1 proto>
      DONEOK                            <T/F><S1 proto>
      IF                                Does S1 relation exist?
        DUP                             <S1 proto><S1 proto>
        @@SOURCE2                       <S2><S1 proto><S1 proto>
        PROTOTYPE                       <S2 proto><S1 proto><S1 proto>
        DONEOK                          <T/F><S2 pro><S1 pro><S1 pro>
        IF                              Does S1 relation exist?
          =                             <T/F><S1 proto>
          IF                            Do S1 and S2 have same proto?
            @@TARGET                    <TGT><S1 proto>
            CREATEREL                   Create TGT relation
            DONEOK                      <T/F>
            IF                          TGT created successfully?
              @@TARGET                  <TGT>
              ACCESSREL                 <TGT ARL>
              DONEOK                    <T/F><TGT ARL>
              IF                        Could access TGT relation?
                @@SOURCE1               <S1><TGT ARL>
```

```
ACCESSREL                   <S1 ARL><TGT ARL>
DONEOK                      <T/F><S1 ARL><TGT ARL>
IF                          Could access S1 relation?
  @@SOURCE2                 <S2><S1 ARL><TGT ARL>
  ACCESSREL                 <S2 ARL><S1 ARL><TGT ARL>
  DONEOK                    <T/F><S2 ARL><S1 ARL><TGT ARL>
  IF                        Could access S2 relation?
    REPEAT                  Loop until EOF <S1 and S2>
      ENDLIST?              <T/F><S2 ARL><S1 ARL><TGT ARL>
      IF                    EOF <S2> ?
        SWAP                <S1 ARL><S2 ARL><TGT ARL>
        ENDLIST?            <T/F><S1 ARL><S2 ARL><TGT ARL>
        NOT                 <F/T><S1 ARL><S2 ARL><TGT ARL>
        IF                  Not EOF <S1>
          ROT               <TGT ARL><S1 ARL><S2 ARL>
          SWAP              <S1 ARL><TGT ARL><S2 ARL>
          REPEAT            Loop through rest of S1
            NEXT_TUPLE      Obtain next tuple in S1
            ENDLIST?        EOF <S1> ?
          UNTIL             Loop through rest of S1
          ROT               <S2 ARL><S1 ARL><TGT ARL>
        THEN
      ELSE                  Not EOF<S2>
        SWAP                <S1 ARL><S2 ARL><TGT ARL>
        ENDLIST?            <T/F><S1 ARL><S2 ARL><TGT ARL>
        IF                  EOF <S1> ?
          SWAP              <S2 ARL><S1 ARL><TGT ARL>
          ROT               <TGT ARL><S2 ARL><S1 ARL>
          SWAP              <S2 ARL><TGT ARL><S1 ARL>
          REPEAT            Loop through rest of S2
            NEXT_TUPLE      Obtain next tuple in S2
            ENDLIST?        EOF <S2> ?
          UNTIL             Loop through rest of S2
          SWAP              <TGT ARL><S2 ARL><S1 ARL>
          -ROT              <S2 ARL><S1 ARL><TGT ARL>
        ELSE                Not EOF<S1> or EOF<S2>
          SWAP              <S2 ARL><S1 ARL><TGT ARL>
          =                 <T/F><S2 ARL><S1 ARL><TGT ARL>
          IF                Is S1 tuple = S2 tuple?
            ROT             <TGT ARL><S2 ARL><S1 ARL>
            COPY_TUPLE      Copy S2 tuple to TGT relation
            -ROT            <S2 ARL><S1 ARL><TGT ARL>
          ELSE              S1 tuple ¬= S2 tuple
            >
            IF              Is S1 tuple > S2 tuple?
              NEXT_TUPLE    Obtain next tuple in S2
            ELSE            S1 tuple < S2 tuple
              SWAP          <S1 ARL><S2 ARL><TGT ARL>
              NEXT_TUPLE    Obtain next tuple in S1
              SWAP          <S2 ARL><S1 ARL><TGT ARL>
          THEN
        THEN
      THEN
      SWAP                  <S1 ARL><S2 ARL><TGT ARL>
      ENDLIST?              EOF <S1> ?
      ROT                   <S2 ARL><T/F><S1 ARL><TGT ARL>
      ENDLIST?              EOF <S2> ?
      ROT                   <T/F><T/F><S2><S1><TGT ARL>
      AND                   <T/F><S2 ARL><S1 ARL><TGT ARL>
    UNTIL                   Loop until EOF <S1 and S2>
    RELEASEREL              Release ARL for S2 relation
  ELSE
    " COULD NOT ACCESS SOURCE RELATION #2
    .
    DROP
  THEN
```

```
                RELEASEREL                  Release ARL for S1 relation
              ELSE
                " COULD NOT ACCESS SOURCE RELATION #1
                .
                DROP
              THEN
              RELEASEREL                    Release ARL for TGT relation
            ELSE
              " COULD NOT ACCESS TARGET RELATION
              .
              DROP
            THEN
          ELSE
            " COULD NOT CREATE TARGET RELATION
            .
            DROP
          THEN
        ELSE
          " SOURCE RELATIONS MUST HAVE SAME PROTOTYPE
          .
          DROP
        THEN
      ELSE
        " SOURCE #2 DOES NOT EXIST
        .
        DROP
      THEN
    ELSE
      " SOURCE #1 DOES NOT EXIST
      .
      DROP
    THEN
  ELSE
    " THE TARGET RELATION ALREADY EXISTS
    .
    DROP
  THEN

  @@SOURCE1                                 Delete constant for S1
  FORGET
  @@SOURCE2                                 Delete constant for S2
  FORGET
  @@TARGET                                  Delete constant for TGT
  FORGET

ELSE
  " INTERSECT COMMAND REQUIRES AT LEAST 3 STACK ELEMENTS
  .
  DROP

END_DEF
```

Command: FINDFLDS

Function: Create a new relation containing copies of all tuples in the source relation having specific field values. The number of field values is variable. All field values must match in order for a tuple to be copied to the new relation.

Stack Before:

Position	Description	Data Type
TOS	Name of target (TGT)	CHAR_ELEM
TOS-1	Name of source (SRC)	CHAR_ELEM
TOS-2	# Fields to compare (#F)	INT_ELEM
TOS-3	Field 1 name (F1)	CHAR_ELEM
TOS-4	Value of Field 1 (V1)	any data type
.	.	.
.	.	.
TOS-x	Field N name (Fn)	CHAR_ELEM
TOS-x-1	Value of Field N (Vn)	any data type
TOS-x-2	<stack element #1>	any data type

Stack After:

Position	Description	Data Type
TOS	<stack element #1>	any data type

In the command below, <Vx> means the desired value of field x (Fx); <V'x> means the actual value of field x.

```
" FINDFLDS
DEFINITION

  #STACK                          <# elements><TGT><SRC><#F><F1>...<Vn>
  3                               <3><# elements><TGT><SRC><#F><F1>..<Vn>
  >=                              <T/F><TGT><SRC><#F><F1>...<Vn>
  IF                              Enough elements to get # fields?
    " @@TARGET                    <"@@TARGET"><TGT><SRC><#F><F1>...<Vn>
    CONSTANT                      Constant for name of TGT relation
    " @@SOURCE                    <"@@SOURCE"><SRC><#F><F1>...<Vn>
    CONSTANT                      Constant for name of SRC relation
    " @@#FLDS                     <"@@#FLDS"><#F><F1>...<Vn>
    CONSTANT                      Constant for # fields to process
    #STACK                        <# elements><F1>...<Vn>
    @@#FLDS                       <#F><# elements><F1>...<Vn>
    2                             <2><#F><# elements><F1>...<Vn>
    *                             <2 * #F><# elements><F1>...<Vn>
    >=                            <T/F><F1>...<Vn>
```

```
IF                          Enough elements to execute command?
  @@TARGET                  <TGT><F1>...<Vn>
  PROTOTYPE                 <TGT proto><F1>...<Vn>
  DONEOK                    <T/F><TGT proto><F1>...<Vn>
  IF                        TGT prototype found?
    " TARGET RELATION ALREADY EXISTS
    .
    DROP
  ELSE                      TGT relation does not exist
    @@SOURCE                <SRC><F1>...<Vn>
    PROTOTYPE               <SRC proto><F1>...<Vn>
    DONEOK                  <T/F><SRC proto><F1>...<Vn>
    IF                      SRC prototype found?
      @@TARGET              <TGT><SRC proto><F1>...<Vn>
      CREATEREL             Create TGT Relation
      DONEOK                <T/F><F1>...<Vn>
      IF                    TGT created successfully?
        @@TARGET            <TGT><F1>...<Vn>
        ACCESSREL           <TGT ARL><F1>...<Vn>
        DONEOK              <T/F><TGT ARL><F1>...<Vn>
        IF                  Could TGT be accessed?
          @@SOURCE          <SRC><TGT ARL><F1>...<Vn>
          ACCESSREL         <SRC ARL><TGT ARL><F1>...<Vn>
          DONEOK            <T/F><SRC ARL><TGT ARL><F1>...<Vn>
          IF                Could SRC be accessed?
            ?ENDLIST        <T/F><SRC ARL><TGT ARL><F1>...<Vn>
            NOT             <F/T><SRC ARL><TGT ARL><F1>...<Vn>
            IF              not EOF<SRC>?
              " @@TEMP      <"@@TEMP"><SRC ARL><TGT ARL><F1>...<Vn>
              VARIABLE      Create Temporary variable for index
              @@TEMP        <addr(@@TEMP)><SRC><TGT><F1>...<Vn>
              3             <3><addr(@@TEMP)><SRC><TGT><F1>...<Vn>
              STORE         <SRC ARL><TGT ARL><F1>...<Vn>
              REPEAT        Repeat until EOF<SRC>
                1           <1=T><SRC ARL><TGT ARL><F1>...<Vn>
                1           <1><T><SRC ARL><TGT ARL><F1>...<Vn>
                @@#FLDS     <#F><1><T><SRC ARL><TGT ARL><F1>...<Vn>
                1           <1><#F><1><T><SRC><TGT><F1>...<Vn>
                DO          Loop through all fields
                  @@TEMP    <addr(@@TEMP)><T><SRC><TGT><F1>...<Vn>
                  FETCH     <index><T><SRC ARL><TGT ARL><F1>...<Vn>
                  <STACK>   <Fx><T><SRC><TGT><F1>...<Vn>
                  VALUE_OF  Actual Value of Field Fx in tuple (V'x)
                  @@TEMP    <addr(@@TEMP)><V'x><T><SRC><TGT><F1>...
                  FETCH     <index><V'x><T><SRC><TGT><F1>...<Vn>
                  2         <2><index><V'x><T><SRC><TGT><F1>...<Vn>
                  +         <2+index><V'x><T><SRC><TGT><F1>...<Vn>
                  <STACK>   Desired Value of Field Fx in tuple (Vx)
                  =         <T/F><T><SRC ARL><TGT ARL><F1>...<Vn>
                  AND       AND status of this search & earlier one
                  @@TEMP    <addr(@@TEMP)><T><SRC><TGT><F1>...<Vn>
                  DUP       <addr(@@TEMP)><addr><T><SRC><TGT><F1>..
                  FETCH     <indx><addr(@@TEMP)><T><SRC><TGT><F1>..
                  2         <2><indx><addr><T><SRC><TGT><F1>...<Vn>
                  +         <2+indx><addr><T><SRC><TGT><F1>...<Vn>
                  STORE     <T><SRC ARL><TGT ARL><F1>...<Vn>
                +LOOP       Loop through all fields
                IF          Were all fields found in tuple?
                  SWAP      <TGT ARL><SRC ARL><F1>...<Vn>
                  COPY_TUPLE Copy SRC tuple to TGT
                  SWAP      <SRC ARL><TGT ARL><F1>...<Vn>
                THEN
                NEXT_TUPLE  Obtain next tuple in SRC
                ?ENDLIST    <T/F><SRC ARL><TGT ARL><F1>...<Vn>
              UNTIL         Repeat until EOF<SRC>
            ELSE
```

```
                " SOURCE RELATION IS EMPTY
                .
                DROP
              THEN

              RELEASEREL        Release ARL for SRC relation
            ELSE
              " COULD NOT ACCESS SOURCE RELATION
              .
              DROP
            THEN
            RELEASEREL          Release ARL for TGT relation
            @@#FLDS             <#F><F1>...<Vn>
            2                   <2><#F><F1>...<Vn>
            *                   <2 * #F><F1>...<Vn>
            #DROP               Remove the command arguments
          ELSE
            " COULD NOT ACCESS TARGET RELATION
            .
            DROP
          THEN
        ELSE
          " COULD NOT CREATE TARGET RELATION
          .
          DROP
        THEN
      ELSE
        " COULD NOT DETERMINE SOURCE PROTOTYPE
        .
        DROP
      THEN
    THEN
    " @@TARGET
    FORGET                      Delete constant for TGT name
    " @@SOURCE
    FORGET                      Delete constant for SRC name
    " @@#FLDS
    FORGET                      Delete constant for # Fields
    " @@TEMP
    FORGET                      Delete variable for stack index
  ELSE
    " NOT ENOUGH STACK ELEMENTS TO PERFORM COMMAND
    .
    DROP
  THEN
ELSE
  " NOT ENOUGH STACK ELEMENTS TO PERFORM COMMAND
  .
  DROP
THEN

END_DEF
```

The vita has been removed from the scanned document